dash optimization

Xpress-Mosel
User Guide

Release 1.4

Last update 1 April, 2004

Published by Dash Optimization Ltd

How to Contact Dash

USA, Canada and all Americas

Dash Optimization Inc

Information and Sales: info@dashoptimization.com
Licensing: license-usa@dashoptimization.com
Product Support: support-usa@dashoptimization.com

Tel: +1 (201) 567 9445
Fax: +1 (201) 567 9443

Dash Optimization Inc.
560 Sylvan Avenue
Englewood Cliffs
NJ 07632
USA

Japan

Dash Optimization Japan

Information and Sales: info@jp.dashoptimization.com
Licensing: license@jp.dashoptimization.com
Product Support: support@jp.dashoptimization.com

Tel: +81 43 297 8836
Fax: +81 43 297 8827

WBG Marive-East 21F FASuC B2124
2-6 Nakase Mihama-ku
261-7121 Chiba
Japan

Worldwide

Dash Optimization Ltd

Information and Sales: info@dashoptimization.com
Licensing: license@dashoptimization.com
Product Support: support@dashoptimization.com

Tel: +44 1926 315862
Fax: +44 1926 315854

Quinton Lodge, Binswood Avenue
Leamington Spa
Warwickshire CV32 5RX
UK

For the latest news and Xpress-MP software and documentation updates, please visit the Xpress-MP website at http://www.dashoptimization.com or subscribe to our mailing list.

Contents

I. Using the Mosel language

Introduction

'*Mosel*' is not an acronym. It is pronounced like the German river, mo-zul. It is an advanced modeling and solving language and environment, where optimization problems can be specified and solved with the utmost precision and clarity.

Here are some of the features of Mosel

- Mosel's *easy syntax* is regular and described formally in the reference manual.
- Mosel supports *dynamic objects*, which do not require pre-sizing. For instance, you do not have to specify the maximum sizes of the indices of a variable x.
- Mosel models are *pre-compiled*. Mosel compiles a model into a binary file which can be run on any computer platform, and which hides the intellectual property in the model if so required.
- Mosel is *embeddable*. There is a runtime library which can be called from your favorite programming language if required. You can access any of the model's objects from your programming language.
- Mosel is *easily extended* through the concept of modules It is possible to write a set of functions, which together stand alone as a module. Several modules are supplied by Dash, including the Xpress-MP Optimizer.
- Support for *user-written functions* and procedures is provided.
- The use of *sets of objects* is supported.
- Constraints and variables etc. can be added *incrementally*. For instance, column generation can depend on the results of previous optimizations, so sub problems are supported

The modeling component of Mosel provides you with an easy to use yet powerful language for describing your problem. It enables you to gather the problem data from text files and a range of popular spreadsheets and databases, and gives you access to a variety of solvers, which can find optimal or near-optimal solutions to your model.

What you need to know before using Mosel

Before using Mosel you should be comfortable with the use of symbols such as x or y to represent unknown quantities, and the use of this sort of variable in simple linear equations and inequalities, for example:

$$x + y \leq 6$$

Experience of a basic course in Mathematical or Linear Programming is worthwhile, but is not essential. Similarly some familiarity with the use of computers would be helpful.

For all but the simplest models you should also be familiar with the idea of summing over a range of variables. For example, if $produce_j$ is used to represent the number of cars produced on production line j then the total number of cars produced on all N production lines can be written as:

$$\sum_{j=1}^{N} produce_j$$

This says 'sum the output from each production line $produce_j$ over all production lines j from $j = 1$ to $j = N$'.

If our target is to produce at least 1000 cars in total then we would write the inequality:

$$\sum_{j=1}^{N} produce_j \geq 1000$$

We often also use a set notation for the sums. Assuming that *LINES* is the set of production lines $\{1, .., N\}$, we may write equivalently:

$$\sum_{j \in LINES} produce_j \geq 1000$$

This may be read 'sum the output from each production line $produce_j$ over all production lines j in the set *LINES*'.

Other common mathematical symbols that are used in the text are $\mathbb{N}$ (the set of non-negative integer numbers $\{0, 1, 2, \ldots\}$), $\cap$ and $\cup$ (intersection and union of sets), $\wedge$ and $\vee$ (logical 'and' and 'or'), the all-quantifier $\forall$ (read 'for all'), and $\exists$ (read 'exists').

Mosel closely mimics the mathematical notation an analyst uses to describe a problem. So provided you are happy using the above mathematical notation the step to using a modeling language will be straightforward.

Symbols and conventions

We have used the following conventions within this guide:

- Mathematical objects are presented in *italics*.
- Examples of commands, models and their output are printed in a `Courier font`. Filenames are given in lower case Courier.
- Decision variables have lower case names; in the most example problems these are verbs (such as *use*, *take*).
- Constraint names start with an upper case letter, followed by mostly lower case (*e.g. Profit*, *TotalCost*).
- Data (arrays and sets) and constants are written entirely with upper case (*e.g. DEMAND*, *COST*, *ITEMS*).
- The vertical bar symbol | is found on many keyboards as a vertical line with a small gap in the middle, but often confusingly displays on-screen without the small gap. In the UNIX world it is referred to as the pipe symbol. (Note that this symbol is not the same as the character sometimes used to draw boxes on a PC screen.) In ASCII, the | symbol is 7C in hexadecimal, 124 in decimal.

The structure of this guide

This user guide is structured into these main parts

- **Part I** describes the use of Mosel for people who want to build and solve Mathematical Programming (MP) problems. These will typically be Linear Programming (LP), Mixed Integer Programming (MIP), or Quadratic Programming (QP) problems. The part has been designed to show the modeling aspects of Mosel, omitting most of the more advanced programming constructs.
- **Part II** is designed to help those users who want to use the powerful programming language facilities of Mosel, using Mosel as a modeling, solving and programming environment. Items covered include looping (with examples), more about using sets, producing nicely formatted output, functions and procedures. We also give some advanced MP examples, including Branch-and-Cut, column generation, Goal Programming and Successive Linear Programming.
- **Part III** shows how Mosel models can be embedded into large applications using programming languages like C, Java, or Visual Basic.

This user guide is deliberately informal and is not complete. It must be read in conjunction with the Mosel reference manual, where features are described precisely and completely.

Chapter 1
Getting started with Mosel

In this chapter we will take you through a very small manufacturing example to illustrate the basic building blocks of Mosel.

Models are entered into a Mosel file using a standard text editor (do not use a word processor as an editor as this may not produce an ASCII file). If you have access to Windows, Xpress-IVE is the model development environment to use. The Mosel file is then loaded into Mosel, and compiled. Finally, the compiled file can be run. This chapter will show the stages in action.

1.1 The chess set problem: description

To illustrate the model development and solving process we shall take a very small example.

A joinery makes two different sizes of boxwood chess sets. The smaller size requires 3 hours of machining on a lathe and the larger only requires 2 hours, because it is less intricate. There are four lathes with skilled operators who each work a 40 hour week. The smaller chess set requires 1 kg of boxwood and the larger set requires 3 kg. However boxwood is scarce and only 200 kg per week can be obtained.

When sold, each of the large chess sets yields a profit of $20, and one of the small chess set has a profit of $5. The problem is to decide how many sets of each kind should be made each week to maximize profit.

1.1.1 A first formulation

Within limits, the joinery can **vary** the number of large and small chess sets produced: there are thus two **decision variables** (or simply **variables**) in our model, one decision variable per product. We shall give these variables abbreviated names:

$$\begin{aligned} \mathit{small}: &\quad \text{the number of small chess sets to make} \\ \mathit{large}: &\quad \text{the number of large chess sets to make} \end{aligned}$$

The number of large and small chess sets we should produce to achieve the maximum contribution to profit is determined by the optimization process. In other words, we look to the optimizer to tell us the best values of *small*, and *large*.

The values which small and large can take will always be **constrained** by some physical or technological limits: they may be constrained to be equal to, less than or greater than some constant. In our case we note that the joinery has a maximum of 160 hours of machine time available per week. Three hours are needed to produce each small chess set and two hours are needed to produce each large set. So the number of hours of machine time actually used each week is $3 \cdot \mathit{small} + 2 \cdot \mathit{large}$. One constraint is thus:

$$3 \cdot \mathit{small} + 2 \cdot \mathit{large} \leq 160 \quad \text{(lathe-hours)}$$

which restricts the allowable combinations of small and large chess sets to those that do not exceed the lathe-hours available.

In addition, only 200 kg of boxwood is available each week. Since small sets use 1 kg for every set made, against 3 kg needed to make a large set, a second constraint is:

$$1 \cdot small + 3 \cdot large \leq 200 \quad \text{(kg of boxwood)}$$

where the left hand side of the inequality is the amount of boxwood we are planning to use and the right hand side is the amount available.

The joinery cannot produce a negative number of chess sets, so two further **non-negativity constraints** are:

$$small \geq 0$$
$$large \geq 0$$

In a similar way, we can write down an expression for the total profit. Recall that for each of the large chess sets we make and sell we get a profit of $20, and one of the small chess set gives us a profit of $5. The total profit is the sum of the individual profits from making and selling the *small* small sets and the *large* large sets, *i.e.*

$$Profit = 5 \cdot small + 20 \cdot large$$

Profit is the **objective function**, a linear function which is to be optimized, that is, maximized. In this case it involves all of the decision variables but sometimes it involves just a subset of the decision variables. In maximization problems the objective function usually represents profit, turnover, output, sales, market share, employment levels or other 'good things'. In minimization problems the objective function describes things like total costs, disruption to services due to breakdowns, or other less desirable process outcomes.

The collection of variables, constraints and objective function that we have defined are our **model**. It has the form of a **Linear Programming problem**: all constraints are linear equations or inequalities, the objective function also is a linear expression, and the variables may take any non-negative real value.

1.2 Solving the chess set problem

1.2.1 Building the model

The Chess Set problem can be solved easily using Mosel. The first stage is to get the model we have just developed into the syntax of the Mosel language. Remember that we use the notation that items in italics (for example, *small*) are the mathematical variables. The corresponding Mosel variables will be the same name in non-italic courier (for example, `small`).

We illustrate this simple example by using the command line version of Mosel. The model can be entered into a file named, perhaps, `chess.mos` as follows:

```
model "Chess"
 declarations
  small: mpvar                      ! Number of small chess sets to make
  large: mpvar                      ! Number of large chess sets to make
 end-declarations

 Profit:=  5*small + 20*large        ! Objective function
 Lathe:=   3*small + 2*large <= 160  ! Lathe-hours
 Boxwood:=   small + 3*large <= 200  ! kg of boxwood
end-model
```

Indentations are purely for clarity. The symbol `!` signifies the start of a **comment**, which continues to the end of the line. Comments over multiple lines start with `(!` and terminate with `!)`.

Notice that the character '*' is used to denote multiplication of the decision variables by the units of machine time and wood that one unit of each uses in the Lathe and Boxwood constraints.

The modeling language distinguishes between upper and lower case, so Small would be recognized as different from small.

Let's see what this all means.

A **model** is enclosed in a model/end-model block.

The **decision variables** are declared as such in the declarations/end-declarations block. Every decision variable must be declared. LP, MIP and QP variables are of type mpvar. Several decision variables can be declared on the same line, so

```
declarations
 small, large: mpvar
end-declarations
```

is exactly equivalent to what we first did. By default, Mosel assumes that all mpvar variables are constrained to be non-negative unless it is informed otherwise, so there is no need to specify non-negativity constraints on variables.

Here is an example of a **constraint**:

```
Lathe:=   3*small + 2*large <= 160
```

The name of the constraint is Lathe. The actual constraint then follows. If the 'constraint' is unconstrained (for example, it might be an **objective function**), then there is no <=, >= or = part.

In Mosel you enter the entire model before starting to compile and run it. Any errors will be signaled when you try to compile the model, or later when you run it (see Chapter 6 on correcting syntax errors).

1.2.2 Obtaining a solution using Mosel

So far, we have just specified a model to Mosel. Next we shall try to solve it. The first thing to do is to specify to Mosel that it is to use Xpress-Optimizer to solve the problem. Then, assuming we can solve the problem, we want to print out the optimum values of the decision variables, small and large, and the value of the objective function. The model becomes

```
model "Chess 2"
 uses "mmxprs"                     ! We shall use Xpress-Optimizer

 declarations
  small,large: mpvar               ! Decision variables: produced quantities
 end-declarations

 Profit:=  5*small + 20*large        ! Objective function
 Lathe:=   3*small + 2*large <= 160  ! Lathe-hours
 Boxwood:=   small + 3*large <= 200  ! kg of boxwood

 maximize(Profit)                  ! Solve the problem

 writeln("Make ", getsol(small), " small sets")
 writeln("Make ", getsol(large), " large sets")
 writeln("Best profit is ", getobjval)
end-model
```

The line

```
 uses "mmxprs"
```

tells Mosel that Xpress-Optimizer will be used to solve the LP. The Mosel modules mmxprs module provides us with such things as maximization, handling bases *etc.*

The line

```
maximize(Profit)
```

tells Mosel to maximize the objective function called `Profit`.

More complicated are the `writeln` statements, though it is actually quite easy to see what they do. If some text is in quotation marks, then it is written literally. `getsol` and `getobjval` are special Mosel functions that return respectively the optimal value of the argument, and the optimal objective function value. `writeln` writes a line terminator after writing all its arguments (to continue writing on the same line, use `write` instead). `writeln` can take many arguments. The statement

```
writeln("small: ", getsol(small), " large: ", getsol(large) )
```

will result in the values being printed all on one line.

1.2.3 Running Mosel from a command line

When you have entered the complete model into a file (let us call it `chess.mos`), we can proceed to get the solution to our problem. Three stages are required:

1. Compiling `chess.mos` to a compiled file, `chess.bim`
2. Loading the compiled file `chess.bim`
3. Running the model we have just loaded.

We start Mosel at the command prompt, and type the following sequence of commands

```
mosel
compile chess
load chess
run
quit
```

which will compile, load and run the model. We will see output something like that below, where we have highlighted Mosel's output in bold face.

```
mosel
** Xpress-Mosel **
(c) Copyright Dash Associates 1998-2002
> compile chess
Compiling 'chess'...
> load chess
> run
Make 0 small sets
Make 66.6667 large sets
Best profit is 1333.33
Returned value:  0
> quit
Exiting.
```

Since the compile/load/run sequence is so often used, it can be abbreviated to

```
cl chess run
```

or simply

```
exec chess
```

The same steps may be done immediately from the command line:

```
mosel -c "cl chess; run"
```

or

```
mosel -c "exec chess"
```

The `-c` option is followed by a list of commands enclosed in double quotes. With Mosel's silent (`-s`) option

```
mosel -s -c "exec chess"
```

the only output is

```
Make 0 small sets
Make 66.6667 large sets
Best profit is 1333.33
```

1.2.4 Using Xpress-IVE

Under Microsoft Windows you may also use Xpress-IVE, sometimes called just IVE, the Xpress **I**nteractive **V**isual **E**nvironment, for working with your Mosel models. Xpress-IVE is a complete modeling and optimization development environment that presents Mosel in an easy-to-use Graphical User Interface (GUI), with a built-in text editor.

To execute the model file `chess.mos` you need to carry out the following steps.

- Start up IVE.
- Open the model file by choosing *File* > *Open*. The model source is then displayed in the central window (the **IVE Editor**).
- Click the *Run* button (green triangle) or alternatively, choose *Build* > *Run*.

The **Build** pane at the bottom of the workspace is automatically displayed when compilation starts. If syntax errors are found in the model, they are displayed here, with details of the line and character position where the error was detected and a description of the problem, if available. Clicking on the error takes the user to the offending line.

When a model is run, the **Output/Input** pane at the right hand side of the workspace window is selected to display program output. Any output generated by the model is sent to this window. IVE will also provide graphical representations of how the solution is obtained, which are generated by default whenever a problem is optimized. The right hand window contains a number of panes for this purpose, dependent on the type of problem solved and the particular algorithm used. IVE also allows the user to draw graphs by embedding subroutines in Mosel models (see the documentation on the website for further detail).

IVE makes all information about the solution available through the **Entities** pane in the left hand window. By expanding the list of decision variables in this pane and hovering over one with the mouse pointer, its solution and reduced cost are displayed. Dual and slack values for constraints may also be obtained.

Chapter 2

Some illustrative examples

This chapter develops the basics of modeling set out in Chapter 1. It presents some further examples of the use of Mosel and introduces new features:

- **Use of subsripts:** Almost all models of any size have subscripted variables. We show how to define arrays of data and decision variables, introduce the different types of sets that may be used as index sets for these arrays, and also simple loops over these sets.
- **Working with data files:** Mosel provides facilities to read from and write to data files in text format and also from other data sources (databases and spreadsheets).

2.1 The burglar problem

A burglar sees 8 items, of different worths and weights. He wants to take the items of greatest total value whose total weight is not more than the maximum *WTMAX* he can carry.

2.1.1 Model formulation

We introduce binary variables $take_i$ for all i in the set of all items (*ITEMS*) to represent the decision whether item i is taken or not. $take_i$ has the value 1 if item i is taken and 0 otherwise. Furthermore, let $VALUE_i$ be the value of item i and $WEIGHT_i$ its weight. A mathematical formulation of the problem is then given by:

$$\text{maximize} \sum_{i \in ITEMS} VALUE_i \cdot take_i$$

$$\sum_{i \in ITEMS} WEIGHT_i \cdot take_i \leq WTMAX \quad \text{(weight restriction)}$$

$$\forall i \in ITEMS : take_i \in \{0, 1\}$$

The objective function is to maximize the total value, that is, the sum of the values of all items taken. The only constraint in this problem is the weight restriction. This problem is an example of a **knapsack problem**.

2.1.2 Implementation

It may be implemented with Mosel as follows:

```
model Burglar
 uses "mmxprs"

 declarations
  WTMAX = 102                       ! Maximum weight allowed
```

```
  ITEMS = 1..8                   ! Index range for items

  VALUE: array(ITEMS) of real    ! Value of items
  WEIGHT: array(ITEMS) of real   ! Weight of items

  take: array(ITEMS) of mpvar    ! 1 if we take item i; 0 otherwise
 end-declarations

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

! Objective: maximize total value
 MaxVal:= sum(i in ITEMS) VALUE(i)*take(i)

! Weight restriction
 sum(i in ITEMS) WEIGHT(i)*take(i) <= WTMAX

! All variables are 0/1
 forall(i in ITEMS) take(i) is_binary

 maximize(MaxVal)                 ! Solve the MIP-problem

! Print out the solution
 writeln("Solution:\n Objective: ", getobjval)
 forall(i in ITEMS)  writeln(" take(", i, "): ", getsol(take(i)))
end-model
```

When running this model we get the following output:

```
Solution:
 Objective: 280
 take(1): 1
 take(2): 1
 take(3): 1
 take(4): 1
 take(5): 0
 take(6): 1
 take(7): 0
 take(8): 0
```

In this model there are a lot of new features, which we shall now explain.

- **Constants:**

  ```
  WTMAX=102
  ```

  declares a constant called WTMAX, and gives it the value 102. Since 102 is an integer, WTMAX is an integer constant. Anything that is given a value in a declarations block is a constant.

- **Ranges:**

  ```
  ITEMS = 1..8
  ```

  defines a **range set**, that is, a set of consecutive integers from 1 to 8. This range is used as an **index set** for the data arrays (VALUE and WEIGHT) and for the array of decision variables take.

- **Arrays**:

  ```
  VALUE: array(ITEMS) of real
  ```

  defines a one-dimensional array of real values indexed by the range ITEMS. Exactly equivalent would be

  ```
  VALUE: array(1..8) of real    ! Value of items
  ```

Multi-dimensional arrays are declared in the obvious way *e.g.*

```
VAL3: array(ITEMS, 1..20, ITEMS) of real
```

declares a 3-dimensional real array. Arrays of decision variables (type `mpvar`) are declared likewise, as shown in our example:

```
x: array(ITEMS) of mpvar
```

declares an array of decision variables `take(1)`, `take(2)`, ..., `take(8)`.

All objects (scalars and arrays) declared in Mosel are always initialized with a default value:

`real`, `integer`: 0
`boolean`: `false`
`string`: `''` (*i.e.* the empty string)

In Mosel, reals are double precision.

- **Assigning values to arrays:** The values of data arrays may either be assigned in the model as we show in the example or initialized from file (see Section 2.2).

  ```
  VALUE := [15, 100, 90, 60, 40, 15, 10,  1]
  ```

  fills the `VALUE` array as follows:
  `VALUE(1)` gets the value 15; `VALUE(2)` gets the value 100; ..., `VALUE(8)` gets the value 1.

  For a 2-dimensional array such as

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

  we might write

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

  which of course is the same as

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

  but much more intuitive. Mosel places the values in the tuple into `EE` 'going across the rows', with the last subscript varying most rapidly. For higher dimensions, the principle is the same.

- **Summations:**

  ```
  MaxVal:= sum(i in Items) VALUE(i)*x(i)
  ```

  defines a linear expression called `MaxVal` as the sum

  $$\sum_{i \in Items} VALUE_i \cdot x_i$$

- **Naming constraints:**

  Optionally, constraints may be named (as in the chess set example). In the remainder of this manual, we shall name constraints only if we need to refer to them at other places in the model. In most examples, only the objective function is named (here `MaxVal`) — to be able to refer to it in the call to the optimization (here `maximize(MaxVal)`).

- **Simple Looping:**

  ```
  forall(i in ITEMS) take(i) is_binary
  ```

  illustrates looping over all values in an index range. Recall that the index range `ITEMS` is 1, ..., 8, so the statement says that `take(1)`, `take(2)`, ..., `take(8)` are all binary variables. There is another example of the use of `forall` at the penultimate line of the model when writing out all the solution values.

- **Integer Programming variable types:**

  To make an mpvar variable, say variable xbinvar, into a binary (0/1) variable, we just have to say

  ```
  xbinvar is_binary
  ```

  To make an mpvar variable an integer variable, *i.e.* one that can only take on integral values in a MIP problem, we would have

  ```
  xintvar is_integer
  ```

2.1.3 The burglar problem revisited

Consider this model:

```
model Burglar2
 uses "mmxprs"

 declarations
  WTMAX = 102                    ! Maximum weight allowed
  ITEMS = {"camera", "necklace", "vase", "picture", "tv", "video",
           "chest", "brick"}     ! Index set for items

  VALUE: array(ITEMS) of real    ! Value of items
  WEIGHT: array(ITEMS) of real   ! Weight of items

  take: array(ITEMS) of mpvar    ! 1 if we take item i; 0 otherwise
 end-declarations

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

! Objective: maximize total value
 MaxVal:= sum(i in ITEMS) VALUE(i)*take(i)

! Weight restriction
 sum(i in ITEMS) WEIGHT(i)*take(i) <= WTMAX

! All variables are 0/1
 forall(i in ITEMS) take(i) is_binary

 maximize(MaxVal)                 ! Solve the MIP-problem

! Print out the solution
 writeln("Solution:\n Objective: ", getobjval)
 forall(i in ITEMS)  writeln(" take(", i, "): ", getsol(take(i)))
end-model
```

What have we changed? The answer is, 'not very much'.

- **String indices:**

  ```
  ITEMS={"camera", "necklace", "vase", "picture", "tv", "video",
         "chest", "brick"}
  ```

  declares that this time ITEMS is a **set of strings**. The indices now take the string values 'camera', 'necklace' *etc.*

  If we run the model, we get

  ```
  Solution:
   Objective: 280
   x(camera): 1
  ```

```
x(necklace): 1
x(vase): 1
x(picture): 1
x(tv): 0
x(video): 1
x(chest): 0
x(brick): 0
```

- **Continuation lines:**

  Notice that the statement

  ```
  ITEMS={"camera", "necklace", "vase", "picture", "tv", "video",
         "chest", "brick"}
  ```

  was spread over two lines. Mosel is smart enough to recognize that the statement is not complete, so it automatically tries to continue on the next line. If you wish to extend a single statement to another line, just cut it after a symbol that implies a continuation, like an operator (+, -, <=, ...) or a comma (`,`) in order to warn the analyzer that the expression continues in the following line(s). For example

  ```
  ObjMax:= sum(i in Irange, j in Jrange) TAB(i,j) * x(i,j) +
            sum(i in Irange) TIB(i) * delta(i)              +
            sum(j in Jrange) TUB(j) * phi(j)
  ```

  Conversely, it is possible to place several statements on a single line, separating them by semicolons (like `x1 <= 4; x2 >= 7`).

2.2 A blending example

A mining company has two types of ore available: Ore 1 and Ore 2. The ores can be mixed in varying proportions to produce a final product of varying quality. For the product we are interested in, the 'grade' (a measure of quality) of the final product must lie between the specified limits of 4 and 5. It sells for REV = £125 per ton. The costs of the two ores vary, as do their availabilities. The objective is to maximize the total net profit.

2.2.1 Model formulation

Denote the amounts of the ores to be used by use_1 and use_2. Maximizing net profit (*i.e.*, sales revenue less cost $COST_o$ of raw material) gives us the objective function:

$$\sum_{o \in ORES} (REV - COST_o) \cdot use_o$$

We then have to ensure that the grade of the final ore is within certain limits. Assuming the grades of the ores combine linearly, the grade of the final product is:

$$\frac{\sum_{o \in ORES} GRADE_o \cdot use_o}{\sum_{o \in ORES} use_o}$$

This must be greater than or equal to 4 so, cross-multiplying and collecting terms, we have the constraint:

$$\sum_{o \in ORES} (GRADE_o - 4) \cdot use_o \geq 0$$

Similarly the grade must not exceed 5.

$$\frac{\sum_{o \in ORES} GRADE_o \cdot use_o}{\sum_{o \in ORES} use_o} \leq 5$$

So we have the further constraint:

$$\sum_{o \in ORES} (5 - GRADE_o) \cdot use_o \geq 0$$

Finally there is a limit to the availability $AVAIL_o$ of each of the ores. We model this with the constraints:

$$\forall o \in ORES : use_o \leq AVAIL_o$$

2.2.2 Implementation

The above problem description sets out the relationships which exist between variables but contains few explicit numbers. Focusing on relationships rather than figures makes the model much more flexible. In this example only the selling price *REV* and the upper/lower limits on the grade of the final product (*MINGRADE* and *MAXGRADE*) are fixed.

Enter the following model into a file `blend.mos`.

```
model "Blend"
 uses "mmxprs"

 declarations
  REV = 125                      ! Unit revenue of product
  MINGRADE = 4                   ! Minimum permitted grade of product
  MAXGRADE = 5                   ! Maximum permitted grade of product
  ORES = 1..2                    ! Range of ores

  COST: array(ORES) of real      ! Unit cost of ores
  AVAIL: array(ORES) of real     ! Availability of ores
  GRADE: array(ORES) of real     ! Grade of ores (measured per unit of mass)

  use: array(ORES) of mpvar      ! Quantities of ores used
 end-declarations

! Read data from file blend.dat
 initializations from 'blend.dat'
  COST
  AVAIL
  GRADE
 end-initializations

! Objective: maximize total profit
 Profit:= sum(o in ORES) (REV-COST(o))* use(o)

! Lower and upper bounds on ore quality
 sum(o in ORES) (GRADE(o)-MINGRADE)*use(o) >= 0
 sum(o in ORES) (MAXGRADE-GRADE(o))*use(o) >= 0

! Set upper bounds on variables
 forall(o in ORES) use(o) <= AVAIL(o)

 maximize(Profit)                 ! Solve the LP-problem

 ! Print out the solution
 writeln("Solution:\n Objective: ", getobjval)
 forall(o in ORES)  writeln(" use(" + o + "): ", getsol(use(o)))

end-model
```

The file `blend.dat` contains the following:

```
! Data file for 'blend.mos'
COST: [85 93]
AVAIL: [60 45]
GRADE: [2.1 6.3]
```

The `initializations from`/`end-initializations` block is new here, telling Mosel where to get data from to initialize named arrays. The order of the data items in the file does not have to be the same as that in the initializations block; equally acceptable would have been the statements

```
initializations from 'blend.dat'
 AVAIL GRADE COST
end-initializations
```

Alternatively, since all data arrays have the same indices, they may be given in the form of a single record, such as BLENDDATA in the following data file `blendb.dat`:

```
        !   [COST AVAIL GRADE]
BLENDDATA: [ [85   60   2.1]
             [93   45   6.3] ]
```

In the `initializations` block we need to indicate the label of the data record and in which order the data of the three arrays is given:

```
initializations from 'blendb.dat'
 [COST,AVAIL,GRADE] as 'BLENDDATA'
end-initializations
```

2.2.3 Re-running the model with new data

There is a problem with the model we have just presented — the name of the file containing the costs date is hard-wired into the model. If we wanted to use a different file, say `blend2.dat`, then we would have to edit the model, and recompile it.

Mosel has **parameters** to help with this situation. A model parameter is a symbol, the value of which can be set just before running the model, often as an argument of the `run` command of the command line interpreter.

```
model "Blend 2"
 uses "mmxprs"

 parameters
  DATAFILE="blend.dat"
 end-parameters

 declarations
  REV = 125                     ! Unit revenue of product
  MINGRADE = 4                  ! Minimum permitted grade of product
  MAXGRADE = 5                  ! Maximum permitted grade of product
  ORES = 1..2                   ! Range of ores

  COST: array(ORES) of real     ! Unit cost of ores
  AVAIL: array(ORES) of real    ! Availability of ores
  GRADE: array(ORES) of real    ! Grade of ores (measured per unit of mass)

  use: array(ORES) of mpvar     ! Quantities of ores used
 end-declarations

! Read data from file
 initializations from DATAFILE
  COST
  AVAIL
```

```
  GRADE
 end-initializations

 ...

end-model
```

The parameter DATAFILE is recognized as a string, and its default value is specified. If we have previously compiled the model into say blend2.bim, then the command

```
mosel -c "load blend2; run 'DATAFILE="blend2.dat"'"
```

will read the cost data from the file we want. Or to compile, load, and run the model using a single command:

```
mosel -c "exec blend2 'DATAFILE="blend2.dat"'"
```

2.2.4 Reading data from spreadsheets and databases

It is quite easy to create and maintain data tables in text files but in many industrial applications data are provided in the form of spreadsheets or need to be extracted from databases. So there is a facility in Mosel whereby the contents of ranges within spreadsheets may be read into data tables and databases may be accessed. It requires an additional authorization in your Xpress-MP license.

On the Dash website, separate documentation is provided for the SQL/ODBC interface (Mosel module mmodbc). To give you a flavor of how Mosel's SQL interface may be used, we now read the data of the blending problem from a spreadsheet and then later from a database.

2.2.4.1 Spreadsheet example

Let us suppose that in a Microsoft Excel spreadsheet called blend.xls you have inserted the following into the cells indicated:

Table 2.1: Spreadsheet example data

	A	B	C	D	E	F
1						
2		ORES	COST	AVAIL	GRADE	
3		1	85	60	2.1	
4		2	93	45	6.3	
5						

and called the range B2:E4 MyRange.

In Windows you need to set up a *User Data Source* called Excel Files, 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.

The following model reads the data for the arrays COST, AVAIL, and GRADE from the Excel range MyRange. Note that we have added "mmodbc" to the uses statement to indicate that we are using the Mosel SQL/ODBC module.

```
model "Blend 3"
 uses  "mmodbc", "mmxprs"

 declarations
  REV = 125                      ! Unit revenue of product
  MINGRADE = 4                   ! Minimum permitted grade of product
  MAXGRADE = 5                   ! Maximum permitted grade of product
  ORES = 1..2                    ! Range of ores

  COST: array(ORES) of real      ! Unit cost of ores
```

```
  AVAIL: array(ORES) of real     ! Availability of ores
  GRADE: array(ORES) of real     ! Grade of ores (measured per unit of mass)

  use: array(ORES) of mpvar      ! Quantities of ores used
 end-declarations

! Read data from spreadsheet blend.xls
 SQLconnect('DSN=Excel Files; DBQ=blend.xls')
 SQLexecute("select * from MyRange ", [COST,AVAIL,GRADE])

 ...

end-model
```

The SQL statement "`select * from MyRange`" says 'select everything from the range called `MyRange`'. It is possible to have much more complex selection statements than the ones we have used.

2.2.4.2 Database example

If we use Microsoft Access, we might have set up an ODBC *DSN* called `MSAccess`, and suppose we are extracting data from a table called `MyTable` in the database `blend.mdb`. There are just the four columns `ORES`, columns `COST`, `AVAIL`, and `GRADE` in `MyTable`, and the data are the same as in the Excel example above. We modify the example above to be

```
model "Blend 4"
 uses  "mmodbc", "mmxprs"

 declarations
  REV = 125                      ! Unit revenue of product
  MINGRADE = 4                   ! Minimum permitted grade of product
  MAXGRADE = 5                   ! Maximum permitted grade of product
  ORES = 1..2                    ! Range of ores

  COST: array(ORES) of real      ! Unit cost of ores
  AVAIL: array(ORES) of real     ! Availability of ores
  GRADE: array(ORES) of real     ! Grade of ores (measured per unit of mass)

  use: array(ORES) of mpvar      ! Quantities of ores used
 end-declarations

! Read data from database blend.mdb
 SQLconnect('DSN=Access; DBQ=blend.mdb')
 SQLexecute("select * from MyTable ", [COST,AVAIL,GRADE])

 ...

end-model
```

To use other databases, for instance a *mysql* database (let us call it `blend`), we merely need to modify the connection string — provided that we have given the same names to the data table and its columns:

```
 SQLconnect('DSN=mysql; DB=blend')
```

ODBC, just like Mosel's text file format, may also be used to output data. The reader is referred to the ODBC/SQL documentation for more detail.

With version 1.4 of Mosel it becomes possible to access data sources through ODBC directly from `initializations` blocks with the same syntax as what has been shown previously for ordinary text format data files. In this case the SQL commands for reading or writing data are generated by Mosel. The whitepaper *Generalized file handling in Mosel* gives several examples of this simplified use of ODBC.

Chapter 3

More advanced modeling features

This chapter introduces some more advanced features of the modeling language in Mosel. We shall not attempt to cover all its features or give the detailed specification of their formats. These are covered in greater depth in the Mosel Reference Manual.

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

The main areas covered in this chapter are related to this property:

- dynamic arrays
- sparse data
- conditional generation
- displaying data

We start again with an example problem. The following sections deal with the different topics in more detail.

3.1 A transport example

A company produces the same product at different plants in the UK. Every plant has a different production cost per unit and a limited total capacity. The customers (grouped into customer regions) may receive the product from different production locations. The transport cost is proportional to the distance between plants and customers, and the capacity on every delivery route is limited. The objective is to minimize the total cost, whilst satisfying the demands of all customers.

3.1.1 Model formulation

Let *PLANT* be the set of plants and *REGION* the set of customer regions. We define decision variables $flow_{pr}$ for the quantity transported from plant p to customer region r. The total cost of the amount of product p delivered to region r is given as the sum of the transport cost (the distance between p and r multiplied by a factor *FUELCOST*) and the production cost at plant p:

$$\text{minimize} \sum_{p \in PLANT} \sum_{r \in REGION} (FUELCOST \cdot DISTANCE_{pr} + PLANTCOST_p) \cdot flow_{pr}$$

The limits on plant capacity are give through the constraints

$$\forall p \in PLANT : \sum_{r \in REGION} flow_{pr} \leq PLANTCAP_p$$

We want to meet all customer demands:

$$\forall r \in REGION : \sum_{p \in PLANT} flow_{pr} = DEMAND_r$$

The transport capacities on all routes are limited:

$$\forall p \in PLANT, r \in REGION : flow_{pr} \leq TRANSCAP_{pr}$$

For simplicity's sake, in this mathematical model we assume that all routes $p \rightarrow r$ are defined and that we have $TRANSCAP_{pr} = 0$ to indicate that a route cannot be used.

3.1.2 Implementation

This problem may be implemented with Mosel as shown in the following:

```
model Transport
 uses "mmxprs"

 declarations
  REGION: set of string                 ! Set of customer regions
  PLANT: set of string                  ! Set of plants

  DEMAND: array(REGION) of real         ! Demand at regions
  PLANTCAP: array(PLANT) of real        ! Production capacity at plants
  PLANTCOST: array(PLANT) of real       ! Unit production cost at plants
  TRANSCAP: array(PLANT,REGION) of real ! Capacity on each route plant->region
  DISTANCE: array(PLANT,REGION) of real ! Distance of each route plant->region
  FUELCOST: real                        ! Fuel cost per unit distance

  flow: array(PLANT,REGION) of mpvar    ! Flow on each route
 end-declarations

 initializations from 'transprt.dat'
  DEMAND
  [PLANTCAP,PLANTCOST] as 'PLANTDATA'
  [DISTANCE,TRANSCAP] as 'ROUTES'
  FUELCOST
 end-initializations

! Create the flow variables that exist
 forall(p in PLANT, r in REGION | exists(TRANSCAP(p,r)) ) create(flow(p,r))

! Objective: minimize total cost
 MinCost:= sum(p in PLANT, r in REGION | exists(flow(p,r)))
            (FUELCOST * DISTANCE(p,r) + PLANTCOST(p)) * flow(p,r)

! Limits on plant capacity
 forall(p in PLANT) sum(r in REGION) flow(p,r) <= PLANTCAP(p)

! Satisfy all demands
 forall(r in REGION) sum(p in PLANT) flow(p,r) = DEMAND(r)

! Bounds on flows
 forall(p in PLANT, r in REGION | exists(flow(p,r)))
  flow(p,r) <= TRANSCAP(p,r)
```

```
 minimize(MinCost)                        ! Solve the problem

end-model
```

REGION and PLANT are declared to be sets of strings, as yet of unknown size. The data arrays (DEMAND, PLANTCAP, PLANTCOST, TRANSCAP, and DISTANCE) and the array of variables flow are indexed by members of REGION and PLANT, their size is therefore not known at their declaration: they are created as **dynamic arrays**. There is a slight difference between dynamic arrays of data and of decision variables (type mpvar): an entry of a data array is created automatically when it is used in the Mosel program, entries of decision variable arrays need to be created explicitly (see Section 3.2.1 below).

The data file transprt.dat contains the problem specific data. It might have, for instance,

```
DEMAND: [ (Scotland) 2840 (North) 2800 (SWest) 2600 (SEast) 2820 (Midlands) 2750]

                      ! [CAP  COST]
PLANTDATA: [ (Corby)    [3000 1700]
             (Deeside) [2700 1600]
             (Glasgow) [4500 2000]
             (Oxford)  [4000 2100] ]

                            ! [DIST CAP]
ROUTES: [ (Corby   North)    [400 1000]
          (Corby   SWest)    [400 1000]
          (Corby   SEast)    [300 1000]
          (Corby   Midlands) [100 2000]
          (Deeside Scotland) [500 1000]
          (Deeside North)    [200 2000]
          (Deeside SWest)    [200 1000]
          (Deeside SEast)    [200 1000]
          (Deeside Midlands) [400  300]
          (Glasgow Scotland) [200 3000]
          (Glasgow North)    [400 2000]
          (Glasgow SWest)    [500 1000]
          (Glasgow SEast)    [900  200]
          (Oxford  Scotland) [800    *]
          (Oxford  North)    [600 2000]
          (Oxford  SWest)    [300 2000]
          (Oxford  SEast)    [200 2000]
          (Oxford  Midlands) [400  500] ]

FUELCOST: 17
```

where we give the ROUTES data only for possible plant/region routes, indexed by the plant and region. It is possible that some data are not specified; for instance, there is no Corby $\leftrightarrow$ Scotland route. So the data are **sparse** and we just create the flow variables for the routes that exist. (The '*' for the (Oxford,Scotland) entry in the capacity column indicates that the entry does not exist; we may write '0' instead: in this case the corresponding *flow* variable will be created but bounded to be 0 by the transport capacity limit).

The **condition** whether an entry in a data table is defined is tested with the Mosel function exists. With the help of the '|' operator we add this test to the forall loop creating the variables. It is not required to add this test to the sums over these variables: only the $flow_{pr}$ variables that have been created are taken into account. However, if the sums involve exactly the index sets that have been used in the declaration of the variables (here this is the case for the objective function MinCost), adding the existence test helps to speed up the enumeration of the existing index-tuples. The following section introduces the conditional generation in a more systematic way.

3.2 Conditional generation — the | operator

Suppose we wish to apply an upper bound to some but not all members of a set of variables x_i. There are *MAXI* members of the set. The upper bound to be applied to x_i is U_i, but it is only to be applied if the entry in the data table TAB_i is greater than 20. If the bound did not depend on the value in TAB_i then the statement would read:

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

Requiring the condition leads us to write

```
forall(i in 1..MAXI | TAB(i) > 20 ) x(i) <= U(i)
```

The symbol '|' can be read as 'such that' or 'subject to'.

Now suppose that we wish to model the following

$$\sum_{\substack{i=1\\A_i>20}}^{MAXI} x_i \leq 15$$

In other words, we just want to include in a sum those x_i for which A_i is greater than 20. This is accomplished by

```
CC:= sum((i in 1..MAXI | A(i)>20 ) x(i) <= 15
```

3.2.1 Conditional variable creation and `create`

As we have already seen in the transport example (Section 3.1), with Mosel we can conditionally create variables. In this section we show a few more examples.

Suppose that we have a set of decision variables `x(i)` where we do not know the set of `i` for which `x(i)` exist until we have read data into an array `WHICH`.

```
model doesx
 declarations
  IR = 1..15
  WHICH: set of integer
  x: dynamic array(IR) of mpvar
 end-declarations

! Read data from file
 initializations from 'doesx.dat'
  WHICH
 end-initializations

! Create the x variables that exist
 forall(i in WHICH) create(x(i))

! Build a little model to show what esists
 Obj:= sum(i in IR) x(i)
 C:= sum(i in IR) i * x(i) >= 5

 exportprob(0, "", Obj)              ! Display the model
end-model
```

If the data in `doesx.dat` are

```
WHICH: [1 4 7 11 14]
```

the output from the model is

```
Minimize
 x(1) + x(4) + x(7) + x(11) + x(14)
Subject To
C: x(1) + 4 x(4) + 7 x(7) + 11 x(11) + 14 x(14) >= 5
Bounds
End
```

Note: `exportprob(0, "", Obj)` is a nice idiom for seeing on-screen the problem that has been created.

The key point is that x has been declared as a **dynamic array**, and then the variables that exist have been created explicitly with `create`. In the transport example in Section 3.1 we have seen a different way of declaring dynamic arrays: the arrays are implicitly declared as dynamic arrays since the index sets are unknown at their declaration.

When we later take operations over the index set of x (for instance, summing), we only include those x that have been created.

Another way to do this, is

```
model doesx2
 declarations
  WHICH: set of integer
 end-declarations

 initializations from 'doesx.dat'
  WHICH
 end-initializations

 finalize(WHICH)

 declarations
  x: array(WHICH) of mpvar        ! Here the array is _not_ dynamic
 end-declarations                 !  because the set has been finalized

 Obj:= sum(i in WHICH) x(i)
 C:= sum(i in WHICH) i * x(i) >= 5

 exportprob(0, "", Obj)
end-model
```

By default, an array is of fixed size if all of its indexing sets are of fixed size (*i.e.* they are either constant or have been **finalized**). Finalizing turns a dynamic set into a constant set consisting of the elements that are currently in the set. All subsequently declared arrays that are indexed by this set will be created as **static** (= fixed size). The second method has two advantages: it is more efficient, and it does not require us to think of the limits of the range `IR` *a priori*.

3.3 Reading sparse data

Suppose we want to read in data of the form

i , j, $value_{ij}$

from an ASCII file, setting up a dynamic array `A(range, range)` with just the `A(i,j)= value_`*ij* for the pairs `(i,j)` which exist in the file. Here is an example which shows three different ways of doing this. We read data from differently formatted files into three different arrays, and using `writeln` show that the arrays hold identical data.

The first method, using the `initializations` block, has already been introduced (transport problem in Section 3.1). The second way of setting up and accessing data demonstrates the immense flexibility of `readln`. As a third possibility, one may use the `diskdata` subroutine from module `mmetc` to read in comma separated value (CSV) files.

```
model "Trio input"
 uses "mmetc"                      ! Required for diskdata

 declarations
  A1, A2, A3: array(range,range) of real
  i, j: integer
 end-declarations

! First method: use an initializations block
 initializations from 'data_1.dat'
  A1 as 'MYDATA'
 end-initializations

! Second method: use the built-in readln function
 fopen("data_2.dat",F_INPUT)
 repeat
  readln('Tut(',i,'and',j,')=', A2(i,j))
 until getparam("nbread") < 6
 fclose(F_INPUT)

! Third method: use diskdata
 diskdata(ETC_IN+ETC_SPARSE,"data_3.dat", A3)

! Now let us see what we have
 writeln('A1 is: ', A1)
 writeln('A2 is: ', A2)
 writeln('A3 is: ', A3)

end-model
```

The data files could be set up thus:

File `data_1.dat`:

```
MYDATA: [ (1 1) 12.5 (2 3) 5.6 (10 9) -7.1 (3 2) 1 ]
```

File `data_2.dat`:

```
Tut(1 and 1)=12.5
Tut(2 and 3)=5.6
Tut(10 and 9)=-7.1
Tut(3 and 2)=1
```

File `data_3.dat`:

```
1, 1, 12.5 2, 3, 5.6 10,9, -7.1 3, 2, 1
```

When printing any of the three arrays A1, A2, or A3 we get the following output:

```
[(1,1,12.5),(2,3,5.6),(3,2,1),(10,9,-7.1)]
```

Chapter 4

Integer Programming

Though many systems can accurately be modeled as Linear Programs, there are situations where discontinuities are at the very core of the decision making problem. There seem to be three major areas where non-linear facilities are required

- where entities must inherently be selected from a discrete set;
- in modeling logical conditions; and
- in finding the global optimum over functions.

Mosel lets you model these non-linearities using a range of discrete (global) entities and then the Xpress-MP Mixed Integer Programming (MIP) optimizer can be used to find the overall (global) optimum of the problem. Usually the underlying structure is that of a Linear Program, but optimization may be used successfully when the non-linearities are separable into functions of just a few variables.

4.1 Integer Programming entities in Mosel

We shall show how to make variables and sets of variables into global entities by using the following declarations.

```
declarations
 IR = 1..8                       ! Index range
 WEIGHT: array(IR) of real       ! Weight table
 x: array(IR) of mpvar
end-declarations

WEIGHT:= [ 2, 5, 7, 10, 14, 18, 22, 30]
```

Xpress-MP handles the following global entities:

- **Binary variables**: decision variables that can take either the value 0 or the value 1 (do/don't do variables).

  We make a variable, say `x(4)`, binary by

  ```
  x(4) is_binary
  ```

- **Integer variables**: decision variables that can take only integer values.

  We make a variable, say `x(7)`, integer by

  ```
  x(7) is_integer
  ```

- **Partial integer variables**: decision variables that can take integer values up to a specified limit and any value above that limit.

  ```
  x(1) is_partint 5   ! Integer up to 5, then continuous
  ```

- **Semi-continuous variables**: decision variables that can take either the value 0, or a value between some lower limit and upper limit. Semi-continuous variables help model situations where if a variable is to be used at all, it has to be used at some minimum level.

  ```
  x(2) is_semcont 6   ! A 'hole' between 0 and 6, then continuous
  ```

- **Semi-continuous integer variables**: decision variables that can take either the value 0, or an integer value between some lower limit and upper limit. Semi-continuous integer variables help model situations where if a variable is to be used at all, it has to be used at some minimum level, and has to be integer.

  ```
  x(3) is_semint 7    ! A 'hole' between 0 and 7, then integer
  ```

- **Special Ordered Sets of type one (SOS1)**: an ordered set of variables at most one of which can take a non-zero value.

- **Special Ordered Sets of type two (SOS2)**: an ordered set of variables, of which at most two can be non-zero, and if two are non-zero these must be consecutive in their ordering.

  If the coefficients in the `WEIGHT` array determine the ordering of the variables, we might form a SOS1 or SOS2 set `MYSOS` by

  ```
  MYSOS:= sum(i in IRng) WEIGHT(i)*x(i) is_sosX
  ```

  where `is_sosX` is either `is_sos1` for SOS1 sets, or `is_sos2` for SOS2 sets.

  Alternatively, if the set `S` holds the members of the set and the linear constraint `L` contains the set variables' coefficients used in ordering the variables (the so-called **reference row entries**), then we can do thus:

  ```
  makesos1(S,L)
  ```

  with the obvious change for SOS2 sets. This method must be used if the coefficient (here `WEIGHT(i)`) of an intended set member is zero. With `is_sosX` the variable will not appear in the set since it does not appear in the linear expression.

  Another point to note about Special Ordered Sets is that the ordering coefficients must be distinct (or else they are not doing their job of supplying an order!).

The most commonly used entities are **binary variables**, which can be employed to model a whole range of logical conditions. **General integers** are more frequently found where the underlying decision variable really has to take on a whole number value for the optimal solution to make sense. For instance, we might be considering the number of airplanes to charter, where fractions of an airplane are not meaningful and the optimal answer will probably involve so few planes that rounding to the nearest integer may not be satisfactory.

Partial integers provide some computational advantages in problems where it is acceptable to round the LP solution to an integer if the optimal value of a decision variable is quite large, but unacceptable if it is small. **Semi-continuous variables** are useful where, if some variable is to be used, its value must be no less than some minimum amount. If the variable is a **semi-continuous integer variable**, then it has the added restriction that it must be integral too.

Special Ordered Sets of type 1 are often used in modeling choice problems, where we have to select at most one thing from a set of items. The choice may be from such sets as: the time period in which to start a job; one of a finite set of possible sizes for building a factory; which machine type to process a part on. **Special Ordered Sets of type 2** are typically used to model non-linear functions of a variable. They are the natural extension of the concepts of Separable Programming, but when embedded in a Branch-and-Bound code (see below) enable truly global optima to be found, and not just local optima. (A local optimum is a point where all the nearest neighbors are worse than it, but where we have no guarantee that there is not a better point some way away. A global optimum is a point which we know to be the best. In the Himalayas the summit of K2 is a local maximum height, whereas the summit of Everest is the global maximum height).

Theoretically, models that can be built with any of the entities we have listed above can be modeled solely with binary variables. The reason why modern IP systems have some or all of the extra entities is that they often provide significant computational savings in computer time and storage when trying to solve the resulting model. Most books and courses on Integer Programming do not emphasize this point adequately. We have found that careful use of the non-binary global entities often yields very considerable reductions in solution times over ones that just use binary variables.

To illustrate the use of Mosel in modeling Integer Programming problems, a small example follows. The first formulation uses binary variables. This formulation is then modified use Special Ordered Sets.

For the interested reader, an excellent text on Integer Programming is *Integer Programming* by Laurence Wolsey, Wiley Interscience, 1998, ISBN 0-471-28366-5.

4.2 A project planning model

A company has several projects that it must undertake in the next few months. Each project lasts for a given time (its duration) and uses up one resource as soon as it starts. The resource profile is the amount of the resource that is used in the months following the start of the project. For instance, project 1 uses up 3 units of resource in the month it starts, 4 units in its second month, and 2 units in its last month.

The problem is to decide when to start each project, subject to not using more of any resource in a given month than is available. The benefit from the project only starts to accrue when the project has been completed, and then it accrues at BEN_p per month for project p, up to the end of the time horizon. Below, we give a mathematical formulation of the above project planning problem, and then display the Mosel model form.

4.2.1 Model formulation

Let $PROJ$ denote the set of projects and $MONTHS = \{1, \ldots, NM\}$ the set of months to plan for. The data are:

DUR_p	the duration of project p
$RESUSE_{pt}$	the resource usage of project p in its t^{th} month
$RESMAX_m$	the resource available in month m
BEN_p	the benefit per month when project finishes

We introduce the binary decision variables $start_{pm}$ that are 1 if project p starts in month m, and 0 otherwise.

The objective function is obtained by noting that the benefit coming from a project only starts to accrue when the project has finished. If it starts in month m then it finishes in month $m+DUR_p-1$. So, in total, we get the benefit of BEN_p for $NM-(m+DUR_p-1) = NM-m-DUR_p+1$ months. We must consider all the possible projects, and all the starting months that let the project finish before the end of the planning period. For the project to complete it must start no later than month $NM - DUR_p$. Thus the profit is:

$$\sum_{p \in PROJ} \sum_{m=1}^{NM-DUR_p} \left(BEN_p \cdot \left(NM - m - DUR_p + 1\right)\right) \cdot start_{pm}$$

Each project must be done once, so it must start in one of the months 1 to $NM - DUR_p$:

$$\forall p \in PROJ : \sum_{m \in MONTHS} start_{pm} = 1$$

We next need to consider the implications of the limited resource availability each month. Note that if a project p starts in month m it is in its $(k - m + 1)^{th}$ month in month k, and so will be using $RESUSE_{p,k-m+1}$ units of the resource. Adding this up for all projects and all starting months up to and including the particular month k under consideration gives:

$$\forall k \in MONTHS : \sum_{p \in PROJ} \sum_{m=1}^{k} RESUSE_{p,k+1-m} \cdot start_{pm} \leq RESMAX_k$$

Finally we have to specify that the $start_{pm}$ are binary (0 or 1) variables:

$$\forall p \in PROJ, m \in MONTHS : start_{pm} \in \{0, 1\}$$

Note that the start month of a project p is given by:

$$\sum_{m=1}^{NM-DUR_p} m \cdot start_{pm}$$

since if an $start_{pm}$ is 1 the summation picks up the corresponding m.

4.2.2 Implementation

The model as specified to Mosel is as follows:

```
model Pplan
 uses "mmxprs"

 declarations
  PROJ = 1..3                       ! Set of projects
  NM = 6                            ! Time horizon (months)
  MONTHS = 1..NM                    ! Set of time periods (months) to plan for

  DUR: array(PROJ) of integer       ! Duration of project p
  RESUSE: array(PROJ,MONTHS) of integer
                                    ! Res. usage of proj. p in its t'th month
  RESMAX: array(MONTHS) of integer ! Resource available in month m
  BEN: array(PROJ) of real          ! Benefit per month once project finished

  start: array(PROJ,MONTHS) of mpvar  ! 1 if proj p starts in month t, else 0
 end-declarations

 DUR   := [3, 3, 4]
 RESMAX:= [5, 6, 5, 5, 4, 5]
 BEN   := [10.2, 12.3, 11.2]
 RESUSE(1,1):= [3, 4, 2]
 RESUSE(2,1):= [4, 1, 6]
 RESUSE(3,1):= [3, 2, 1, 2] ! Other RESUSE entries are 0 by default

! Objective: Maximize Benefit
! If project p starts in month t, it finishes in month t+DUR(p)-1 and
! contributes a benefit of BEN(p) for the remaining NM-(t+DUR(p)-1) months:
 MaxBen:=
  sum(p in PROJ, m in 1..NM-DUR(p)) (BEN(p)*(NM-m-DUR(p)+1)) * start(p,m)

! Each project starts once and only once:
 forall(p in PROJ) One(p):= sum(m in MONTHS) start(p,m) = 1

! Resource availability:
! A project starting in month m is in its k-m+1'st month in month k:
 forall(k in MONTHS) ResMax(k):=
  sum(p in PROJ, m in 1..k) RESUSE(p,k+1-m)*start(p,m) <= RESMAX(k)

! Make all the start variables binary
 forall(p in PROJ, m in MONTHS) start(p,m) is_binary
```

```
 maximize(MaxBen)                    ! Solve the MIP-problem

 writeln("Solution value is: ", getobjval)
 forall(p in PROJ) writeln( p, " starts in month ",
                            getsol(sum(m in 1..NM-DUR(p)) m*start(p,m)) )
end-model
```

Note that in the solution printout we apply the `getsol` function not to a single variable but to a linear expression.

4.3 The project planning model using Special Ordered Sets

The example can be modified to use Special Ordered Sets of type 1 (SOS1). The $start_{pm}$ variables for a given p form a set of variables which are ordered by m, the month. The ordering is induced by the coefficients of the $start_{pm}$ in the specification of the SOS. For example, $start_{p1}$'s coefficient, 1, is less than $start_{p2}$'s, 2, which in turn is less than $start_{p3}$'s coefficient, and so on The fact that the $start_{pm}$ variables for a given p form a set of variables is specified to Mosel as follows:

```
(! Define SOS-1 sets that ensure that at most one start(p,m) is non-zero
   for each project p. Use month index to order the variables !)

 forall(p in PROJ) XSet(p):= sum(m in MONTHS) m*start(p,m) is_sos1
```

The `is_sos1` specification tells Mosel that `Xset(p)` is a Special Ordered Set of type 1. The linear expression specifies both the set members and the coefficients that order the set members. It says that all the $start_{pm}$ variables for m in the *MONTHS* index range are members of an SOS1 with reference row entries m.

The specification of the $start_{pm}$ as binary variables must now be removed. The binary nature of the $start_{pm}$ is implied by the SOS1 property, since if the $start_{pm}$ must add up to 1 and only one of them can differ from zero, then just one is 1 and the others are 0.

If the two formulations are equivalent why were Special Ordered Sets invented, and why are they useful? The answer lies in the way the reference row gives the search procedure in Integer Programming (IP) good clues as to where the best solution lies. Quite frequently the Linear Programming (LP) problem that is solved as a first approximation to an Integer Program gives an answer where $start_{p1}$ is fractional, say with a value of 0.5, and $start_{p,NM}$ takes on the same fractional value. The IP will say:

"my job is to get variables to 0 or 1. Most of the variables are already there so I will try moving xp1 or xpT. Since the set members must add up to 1.0, one of them will go to 1, and one to 0. So I think that we start the project either in the first month or in the last month."

A much better guess is to see that the $start_{pm}$ are ordered and the LP solution is telling us it looks as if the best month to start is somewhere midway between the first and the last month. When sets are present, the IP can branch on sets of variables. It might well separate the months into those before the middle of the period, and those later. It can then try forcing all the early $start_{pm}$ to 0, and restricting the choice of the one $start_{pm}$ that can be 1 to the later $start_{pm}$. It has this option because it now has the information to 'know' what is an early and what is a late $start_{pm}$, whereas these variables were unordered in the binary formulation.

The power of the set formulation can only really be judged by its effectiveness in solving large, difficult problems. When it is incorporated into a good IP system such as Xpress-MP it is often found to be an order of magnitude better than the equivalent binary formulation for large problems.

Chapter 5

Overview of subroutines and reserved words

There is a range of built-in functions and procedures available in Mosel. They are described fully in the Mosel Language Reference Manual. Here is a summary.

- Accessing solution values: `getsol`, `getact`, `getcoeff`, `getdual`, `getrcost`, `getslack`, `getobjval`
- Arithmetic functions: `arctan`, `cos`, `sin`, `ceil`, `floor`, `round`, `exp`, `ln`, `log`, `sqrt`, `isodd`
- List functions: `maxlist`, `minlist`
- String functions:`strfmt`, `substr`
- Dynamic array handling: `create`, `finalize`
- File handling: `fclose`, `fflush`, `fopen`, `fselect`, `fskipline`, `getfid`, `iseof`, `read`, `readln`
- Accessing control parameters: `getparam`, `setparam`
- Getting information: `getsize`, `gettype`, `getvars`
- Hiding constraints: `sethidden`, `ishidden`
- Miscellaneous functions: `exportprob`, `bittest`, `random`, `setcoeff`, `settype`, `exit`

5.1 Modules

The distribution of Mosel contains several **modules** that add extra functionality to the language.

A full list of the functionality of a module can be obtained by using Mosel's `exam` command, for instance

```
mosel -c "exam mmsystem"
```

In this manual, we always use Xpress-Optimizer as solver. Access to the corresponding optimization functions is provided by the module `mmxprs`.

In the `mmxprs` module are the following useful functions.

- Optimize: `minimize`, `maximize`
- MIP directives: `setmipdir`, `clearmipdir`
- Handling bases: `savebasis`, `loadbasis`, `delbasis`

- Force problem loading: `loadprob`
- Accessing problem status: `getprobstat`
- Deal with bounds: `setlb`, `setub`, `getlb`, `getub`
- Model cut functions: `setmodcut`, `clearmodcut`

For example, here is a nice habit to get into when solving a problem with the Xpress-MP Optimizer.

```
declarations
  status:array({XPRS_OPT,XPRS_UNF,XPRS_INF,XPRS_UNB}) of string
end-declarations

status:=["Optimum found","Unfinished","Infeasible","Unbounded"]
...
minimize(Obj)
writeln(status(getprobstat))
```

In the `mmsystem` module are various useful functions provided by the underlying operating system:

- Delete a file/directory: `fdelete`, `removedir`
- Move a file: `fmove`
- Current working directory: `getcwd`
- Get an environment variable's value: `getenv`
- File status: `getfstat`
- Returns the system status: `getsysstat`
- Time: `gettime`
- Make a directory: `makedir`
- General system call: `system`

Other modules mentioned in this manual are `mmodbc` and `mmetc`.

See the module reference manuals for full details.

5.2 Reserved words

The following words are reserved in Mosel. The upper case versions are also reserved (*i.e.* `AND` and `and` are keywords but not `And`). Do not use them in a model except with their built-in meaning.

`and`, `array`, `as`
`boolean`, `break`
`case`
`declarations`, `div`, `do`, `dynamic`
`elif`, `else`, `end`
`false`, `forall`, `forward`, `from`, `function`
`if`, `in`, `include`, `initialisations`, `initializations`, `integer`, `inter`,
`is_binary`, `is_continuous`, `is_free`, `is_integer`, `is_partint`, `is_semcont`,
`is_semint`, `is_sos1`, `is_sos2`
`linctr`
`max`, `min`, `mod`, `model`, `mpvar`

`next`, `not`
`of`, `options`, `or`
`parameters`, `procedure`, `public`, `prod`
`range`, `real`, `repeat`
`set`, `string`, `sum`
`then`, `to`, `true`
`union`, `until`, `uses`
`while`

Chapter 6

Correcting syntax errors in Mosel

The parser of Mosel is able to detect a large number of errors that may occur when writing a model. In this chapter we shall try to analyze and correct some of these.

If we compile the model

```
model ‘Plenty of errors’
 declarations
  small, large: mpvar
 end-declarations

 Profit=   5*small + 20*large
 Boxwood:=   small + 3*large <= 200
 Lathe:=   3*small + 2*large <= 160

 maximize(Profit)

 writeln("Best profit is ", getobjval
end-model
```

we get the following output:

```
Mosel: E-100 at (1,7) of ‘poerror.mos’: Syntax error before ‘‘’.
Parsing failed.
```

The second line of the output informs us that the compilation has not been executed correctly. The first line tells us exactly the type of the error that has been detected, namely a syntax error with the code `E-100` (where E stands for error) and its location: line 1 character 7. The problem is caused by the apostrophe ‘ (or something preceding it). Indeed, Mosel expects either single or double quotes around the name of the model if the name contains blanks. We therefore replace it by ’ and compile the corrected model, resulting in the following display:

```
Mosel: E-100 at (6,8) of ‘poerror.mos’: Syntax error before ‘=’.
Mosel: W-121 at (6,29) of ‘poerror.mos’: Statement with no effect.
Mosel: E-100 at (10,16) of ‘poerror.mos’: ‘Profit’ is not defined.
Mosel: E-123 at (10,17) of ‘poerror.mos’: ‘maximize’ is not defined.
Mosel: E-100 at (12,37) of ‘poerror.mos’: Syntax error.
Parsing failed.
```

There is a problem with the sign = in the 6^{th} line:

```
Profit=  5*small + 20*large
```

In the model body the equality sign = may only be used in the definition of constraints or in logical expressions. Constraints are linear relations between variables, but `profit` has not been defined as a variable, so the parser detects an error. What we really want, is to assign the linear expression `5*small + 20*large` to `Profit`. For such an assignment we have to use the sign `:=`. Using just = is a very common error.

As a consequence of this error, the linear expression after the equality sign does not have any relevance to the problem that is stated. The parser informs us about this fact in the second line: it has found a statement with no effect. This is not an error that would cause the failure of the compilation taken on its own, but simply a **warning** (marked by the W in the error code W-121) that there may be something to look into. Since Profit has not been defined, it cannot be used in the call to the optimization, hence the third error message.

As we have seen, the second and the third error messages are consequences of the first mistake we have made. Before looking at the last message that has been displayed we recompile the model with the corrected line

```
Profit:=  5*small + 20*large
```

to get rid of all side effects of this error. Unfortunately, we still get a few error messages:

```
Mosel: E-123 at (10,17) of ‘poerror.mos’: ‘maximize’ is not defined.
Mosel: E-100 at (12,37) of ‘poerror.mos’: Syntax error.
```

There is still a problem in line 10; this time it shows up at the very end of the line. Although everything appears to be correct, the parser does not seem to know what to do with maximize. The solution to this enigma is that we have forgotten to load the module mmxprs that provides the optimization function maximize. To tell Mosel that this module is used we need to add the line

```
uses "mmxprs"
```

immediately after the start of the model, before the declarations block. Forgetting to specify mmxprs is another common error. We now have a closer look at line 12 (which has now become line 13 due to the addition of the uses statement). All subroutines called in this line (writeln and getobjval) are provided by Mosel, so there must be yet another problem: we have forgotten to close the parentheses. After adding the closing parenthesis after getobjval the model finally compiles without displaying any errors. If we run it we obtain the desired output:

```
Best profit is 1333.33
Returned value: 0
```

Besides the detection of syntax errors, Mosel may also give some help in finding run time errors. It should only be pointed out here that it is possible to add the flag -g to the compile command to obtain some information about where the error occurred in the program. Also useful is turning on verbose reporting, for instance

```
setparam("XPRS_VERBOSE",true)
setparam("XPRS_LOADNAMES",true)
```

II. Advanced language features

This part takes the reader who wants to use Mosel as a modeling, solving **and** programming environment through its powerful programming language facilities. The following topics, most of which have already shortly been mentioned in the first part, are covered in a more detailed way:

- Selections and loops (Chapter 7)
- Working with sets (Chapter 8)
- Functions and procedures (Chapter 9)
- Output to files and producing formatted output (Chapter 10)

Whilst the first four chapters in this part present pure programming examples, the last two chapters contain some advanced examples of LP and MIP that make use of the programming facilities in Mosel:

- Cut generation (Section 11.1)
- Column generation (Section 11.2)
- Recursion or Successive Linear Pogramming (Section 12.1)
- Goal Programming (Section 12.2)

Chapter 7

Flow control constructs

Flow control constructs are mechanisms for controlling the order of the execution of the actions in a program. In this chapter we are going to have a closer look at two fundamental types of control constructs in Mosel: selections and loops.

Frequently actions in a program need to be repeated a certain number of times, for instance for all possible values of some index or depending on whether a condition is fulfilled or not. This is the purpose of **loops**. Since in practical applications loops are often interwoven with conditions (**selection statements**), these are introduced first.

7.1 Selections

Mosel provides several statements to express a selection between different actions to be taken in a program. The simplest form of a selection is the `if-then` statement:

- **if-then**: 'If a condition holds do something'. For example:

```
if A >= 20 then
 x <= 7
endif
```

For an integer number A and a variable x of type `mpvar`, x is constrained to be less or equal to 7 if A is greater or equal 20.

Note that there may be any number of expressions between `then` and `endif`, not just a single one.

In other cases, it may be necessary to express choices with alternatives.

- **if-then-else**: 'If a condition holds, do this, otherwise do something else'. For example:

```
if A >= 20 then
  x <= 7
 else x >= 35
endif
```

  Here the upper bound 7 is applied to the variable x if the value of A is greater or equal 20, otherwise the lower bound 35 is applied to it.

- **if-then-elif-then-else**: 'If a condition holds do this, otherwise, if a second condition holds do something else *etc.*'

```
if A >= 20 then
 x <= 7
elif A <= 10 then
 x >= 35
else
 x = 0
endif
```

Here the upper bound 7 is applied to the variable x if the value of A is greater or equal 20, and if the value of A is less or equal 10 then the lower bound 35 is applied to x. In all other cases (that is, A is greater than 10 and smaller than 20), x is fixed to 0.

Note that this could also be written using two separate `if-then` statements but it is more efficient to use `if-then-elif-then[-else]` if the cases that are tested are mutually exclusive.

- **case**: 'Depending on the value of an expression do something'.

```
case A of
 -MAX_INT..10 : x >= 35
 20..MAX_INT :  x <= 7
 12, 15 :       x = 1
 else           x = 0
endif
```

Here the upper bound 7 is applied to the variable x if the value of A is greater or equal 20, and the lower bound 35 is applied if the value of A is less or equal 10. In addition, x is fixed to 1 if A has value 12 or 15, and fixed to 0 for all remaining values.

An example for the use of the case statement is given in Section 12.2.

The following example uses the `if-then-elif-then` statement to compute the minimum and the maximum of a set of randomly generated numbers:

```
model Minmax

 declarations
  SNumbers: set of integer
  LB=-1000                      ! Elements of SNumbers must be between LB
  UB=1000                       ! and UB
 end-declarations

                                ! Generate a set of 50 randomly chosen numbers
 forall(i in 1..50)
  SNumbers += {round(random*200)-100}

 writeln("Set: ", SNumbers, " (size: ", getsize(SNumbers), ")")

 minval:=UB
 maxval:=LB
 forall(p in SNumbers)
   if p<minval then
     minval:=p
   elif p>maxval then
     maxval:=p
   end-if

 writeln("Min: ", minval, ", Max: ", maxval)

end-model
```

Instead of writing the loop above, it would of course be possible to use the corresponding operators `min` and `max` provided by Mosel:

```
writeln("Min: ", min(p in SNumbers) p, ", Max: ", max(p in SNumbers) p)
```

It is good programming practice to indent the block of statements in loops or selections as in the preceding example so that it becomes easy to get an overview where the loop or the selection ends. — At the same time this may serve as a control whether the loop or selection has been terminated correctly (*i.e.* no `end-if` or similar key words terminating loops have been left out).

7.2 Loops

Loops group actions that need to be repeated a certain number of times, either for all values of some index or counter (`forall`) or depending on whether a condition is fulfilled or not (`while`, `repeat-until`).

This section presents the complete set of loops available in Mosel, namely `forall`, `forall-do`, `while`, `while-do`, and `repeat-until`.

7.2.1 forall

The `forall` loop repeats a statement or block of statements for all values of an index or counter. If the set of values is given as an interval of integers (`range`), the enumeration starts with the smallest value. For any other type of sets the order of enumeration depends on the current (internal) order of the elements in the set.

The `forall` loop exists in two different versions in Mosel. The inline version of the `forall` loop (*i.e.* looping over a single statement) has already been used repeatedly, for example as in the following loop that constrains variables `x(i)` (i=1,...,10) to be binary.

```
forall(i in 1..10) x(i) is_binary
```

The second version of this loop, `forall-do`, may enclose a block of statements, the end of which is marked by `end-do`.

Note that the indices of a `forall` loop can **not** be modified inside the loop. Furthermore, they must be new objects: a symbol that has been declared cannot be used as index of a `forall` loop.

The following example that calculates all perfect numbers between 1 and a given upper limit combines both types of the `forall` loop. (A number is called **perfect** if the sum of its divisors is equal to the number itself.)

```
model Perfect

 parameters
  LIMIT=100
 end-parameters

 writeln("Perfect numbers between 1 and ", LIMIT, ":")

 forall(p in 1..LIMIT) do
   sumd:=1
   forall(d in 2..p-1)
    if p mod d = 0 then          ! Mosel’s built-in mod operator
      sumd+=d                    ! The same as sum:= sum + d
     end-if
   if p=sumd then
     writeln(p)
   end-if
 end-do

end-model
```

The outer loop encloses several statements, we therefore need to use `forall-do`. The inner loop only applies to a single statement (`if` statement) so that we may use the inline version `forall`.

If run with the default parameter settings, this program computes the solution 1, 6, 28.

7.2.1.1 Multiple indices

The `forall` statement (just like the `sum` operator and any other statement in Mosel that requires index set(s)) may take any number of indices, with values in sets of any basic type or ranges of integer values. If two or more indices have the same set of values as in

```
forall(i in 1..10, j in 1..10) y(i,j) is_binary
```

(where `y(i,j)` are variables of type `mpvar`) the following equivalent short form may be used:

```
forall(i,j in 1..10) y(i,j) is_binary
```

7.2.1.2 Conditional looping

The possibility of adding conditions to a `forall` loop via the '|' symbol has already been mentioned in Chapter 3. Conditions may be applied to one or several indices and the selection statement(s) can be placed accordingly. Take a look at the following example where `A` and `U` are one- and two-dimensional arrays of integers or reals respectively, and `y` a two-dimensional array of decision variables (`mpvar`):

```
forall(i in -10..10, j in 0..5 | A(i) > 20) y(i,j) <= U(i,j)
```

For all `i` from -10 to 10, the upper bound `U(i,j)` is applied to the variable `y(i,j)` if the value of `A(i)` is greater than 20.

The same conditional loop may be reformulated (in an equivalent but usually less efficient way) using the `if` statement:

```
forall(i in -10..10, j in 0..5)
 if A(i) > 20
  y(i,j) <= U(i,j)
end-if
```

If we have a second selection statement on both indices with `B` a two-dimensional array of integers or reals, we may either write

```
forall(i in -10..10, j in 0..5 | A(i) > 20 and B(i,j) <> 0 ) y(i,j) <= U(i,j)
```

or, more efficiently, since the second condition on both indices is only tested if the condition on index `i` holds:

```
forall(i in -10..10 | A(i) > 20, j in 0..5 | B(i,j) <> 0 ) y(i,j) <= U(i,j)
```

7.2.2 `while`

A `while` loop is typically employed if the number of times that the loop needs to be executed is not know beforehand but depends on the evaluation of some condition: a set of statements is repeated while a condition holds. As with `forall`, the `while` statement exists in two versions, an inline version (`while`) and a version (`while-do`) that is to be used with a block of program statements.

The following example computes the largest common divisor of two integer numbers `A` and `B` (that is, the largest number by which both `A` and `B`, can be divided without remainder). Since there is only a single `if-then-else` statement in the `while` loop we could use the inline version of the loop but, for clarity's sake, we have given preference to the `while-do` version that marks where the loop terminates clearly.

```
model Lcdiv1

 declarations
  A,B: integer
 end-declarations

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

```
 readln(A)
 write("  B: ")
 readln(B)

 while (A <> B) do
  if (A>B) then
   A:=A-B
  else B:=B-A
  end-if
 end-do

 writeln("Largest common divisor: ", A)

end-model
```

7.2.3 repeat until

The `repeat-until` structure is similar to the `while` statement with the difference that the actions in the loop are executed once before the termination condition is tested for the first time.

The following example combines the three types of loops (`forall`, `while`, `repeat-until`) that are available in Mosel. It implements a **Shell sort** algorithm for sorting an array of numbers into numerical order. The idea of this algorithm is to first sort, by straight insertion, small groups of numbers. Then several small groups are combined and sorted. This step is repeated until the whole list of numbers is sorted.

The spacings between the numbers of groups sorted on each pass through the data are called the increments. A good choice is the sequence which can be generated by the recurrence $inc_1 = 1,\ inc_{k+1} = 3 \cdot inc_k + 1,\ k = 1, 2, \ldots$

```
model "Shell sort"

 declarations
  N: integer                      ! Size of array ANum
  ANum: array(range) of real      ! Unsorted array of numbers
 end-declarations

 N:=50
 forall(i in 1..N)
  ANum(i):=round(random*100)

 writeln("Given list of numbers (size: ", N, "): ")
 forall(i in 1..N) write(ANum(i), " ")
 writeln

 inc:=1                          ! Determine the starting increment
 repeat
   inc:=3*inc+1
 until (inc>N)

 repeat                          ! Loop over the partial sorts
   inc:=inc div 3
   forall(i in inc+1..N) do      ! Outer loop of straight insertion
     v:=ANum(i)
     j:=i
     while (ANum(j-inc)>v) do    ! Inner loop of straight insertion
       ANum(j):=ANum(j-inc)
       j -= inc
       if j<=inc then break; end-if
     end-do
     ANum(j):= v
   end-do
 until (inc<=1)
```

```
 writeln("Ordered list: ")
 forall(i in 1..N) write(ANum(i), " ")
 writeln

end-model
```

The example introduces a new statement: `break`. It can be used to interrupt one or several loops. In our case it stops the inner `while` loop. Since we are jumping out of a single loop, we could as well write `break 1`. If we wrote `break 3`, the `break` would make the algorithm jump 3 loop levels higher, that is outside of the `repeat-until` loop.

Note that there is no limit to the number of nested levels of loops and/or selections in Mosel.

Chapter 8

Sets

A set collects objects of the same type without establishing an order among them (as is the case with arrays). In Mosel, sets may be defined for all elementary types, that is the basic types (`integer`, `real`, `string`, `boolean`) and the MP types (`mpvar` and `linctr`).

This chapter presents in a more systematic way the different possibilities how sets may be initialized (all of which the reader has already encountered in the examples in the first part), and shows also more advanced ways of working with sets.

8.1 Initializing sets

In the revised formulation of the burglar problem in Chapter 2 and also in the models in Chapter 3 we have already seen different examples for the use of index sets. We recall here the relevant parts of the respective models.

8.1.1 Constant sets

In the Burglar example the index set is assigned directly in the model:

```
declarations
 ITEMS={"camera", "necklace", "vase", "picture", "tv", "video",
        "chest", "brick"}
end-declarations
```

Since in this example the set contents is set in the declarations section, the index set `ITEMS` is a **constant set** (its contents cannot be changed). To declare it as a **dynamic set**, the contents needs to be assigned after its declaration:

```
declarations
 ITEMS: set of string
end-declarations

ITEMS:={"camera", "necklace", "vase", "picture", "tv", "video",
        "chest", "brick"}
```

8.1.2 Set initialization from file, finalized and fixed sets

In Chapter 3 the reader has encountered several examples how the contents of sets may be initialized from data files.

The contents of the set may be read in directly as in the following case:

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

```
 initilizations from 'idata.dat'
  WHICH
 end-initializations
```

Where `idata.dat` contains data in the following format:

```
WHICH: [1 4 7 11 14]
```

Unless a set is constant, arrays that are indexed by this set are created as dynamic arrays. Since in many cases the contents of a set does not change any more after its initialization, Mosel provides the `finalize` statement that turns a (dynamic) set into a constant set. Consider the continuation of the example above:

```
finalize(WHICH)

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

The array of variables `x` will be created as a static array, without the `finalize` statement it would be dynamic since the index set `WHICH` may still be subject to changes. Declaring arrays in the form of static arrays is preferable if the indexing set is known before because this allows Mosel to handle them in a more efficient way.

Index sets may also be initialized indirectly during the initialization of dynamic arrays:

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

 initializations from 'transprt.dat'
  DEMAND
 end-initilizations
```

If file `transprt.dat` contains the data:

```
DEMAND: [(Scotland) 2840 (North) 2800 (West) 2600 (SEast) 2820 (Midlands) 2750]
```

then printing the set `REGION` after the initialization will give the following output:

```
{'Scotland','North','West','SEast','Midlands'}
```

Once a set is used for indexing an array (of data, decision variables *etc.*) it is **fixed**, that is, its elements can no longer be removed, but it may still grow in size.

The indirect initialization of (index) sets is not restricted to the case that data is input from file. In the following example we add an array of variable descriptions to the chess problem introduced in Chapter 1. These descriptions may, for instance, be used for generating a nice output. Since the array `DescrV` and its indexing set `Allvars` are dynamic they grow with each new variable description that is added to `DescrV`.

```
model "Chess 3"
 uses "mmxprs"

 declarations
  Allvars: set of mpvar
  DescrV: array(Allvars) of string
  small, large: mpvar
 end-declarations

 DescrV(small):= "Number of small chess sets"
 DescrV(large):= "Number of large chess sets"

 Profit:=  5*small + 20*large
```

```
 Lathe:=   3*small + 2*large <= 160
 Boxwood:=   small + 3*large <= 200

 maximize(Profit)

 writeln("Solution:\n Objective: ", getobjval)
 writeln(DescrV(small), ": ", getsol(small))
 writeln(DescrV(large), ": ", getsol(large))

end-model
```

The reader may have already remarked another feature that is illustrated by this example: the indexing set `Allvars` is of type `mpvar`. So far only basic types have occurred as index set types but as mentioned earlier, sets in Mosel may be of any elementary type, including the MP types `mpvar` and `linctr`.

8.2 Working with sets

In all examples of sets given so far sets are used for indexing other modeling objects. But they may also be used for different purposes.

The following example demonstrates the use of basic set operations in Mosel: **union** (+), **intersection** (*), and **difference** (-):

```
model "Set example"

 declarations
  Cities={"rome", "bristol", "london", "paris", "liverpool"}
  Ports={"plymouth", "bristol", "glasgow", "london", "calais",
              "liverpool"}
  Capitals={"rome", "london", "paris", "madrid", "berlin"}
 end-declarations

 Places:= Cities+Ports+Capitals        ! Create the union of all 3 sets
 In_all_three:= Cities*Ports*Capitals  ! Create the intersection of all 3 sets
 Cities_not_cap:= Cities-Capitals      ! Create the set of all cities that are
                                       ! not capitals

 writeln("Union of all places: ", Places)
 writeln("Intersection of all three: ", In_all_three)
 writeln("Cities that are not capitals: ", Cities_not_cap)

end-model
```

The output of this example will look as follows:

```
  Union of all places:{‘rome’,‘bristol’,‘london’,‘paris’,‘liverpool’,
  ‘plymouth’,‘bristol’,‘glasgow’,‘calais’,‘liverpool’,‘rome’,‘paris’,
  ‘madrid’,‘berlin’}
  Intersection of all three: {‘london’}
  Cities that are not capitals: {‘bristol’,‘liverpool}
```

Sets in Mosel are indeed a powerful facility for programming as in the following example that calculates all **prime numbers** between 2 and some given limit.

Starting with the smallest one, the algorithm takes every element of a set of numbers `SNumbers` (positive numbers between 2 and some upper limit that may be specified when running the model), adds it to the set of prime numbers `SPrime` and removes the number and all its multiples from the set `SNumbers`.

```
model Prime

 parameters
  LIMIT=100                     ! Search for prime numbers in 2..LIMIT
 end-parameters

 declarations
  SNumbers: set of integer      ! Set of numbers to be checked
  SPrime: set of integer        ! Set of prime numbers
 end-declarations

 SNumbers:={2..LIMIT}

 writeln("Prime numbers between 2 and ", LIMIT, ":")

 n:=2
 repeat
   while (not(n in SNumbers)) n+=1
   SPrime += {n}                ! n is a prime number
   i:=n
   while (i<=LIMIT) do          ! Remove n and all its multiples
     SNumbers-= {i}
     i+=n
   end-do
 until SNumbers={}

 writeln(SPrime)
 writeln(" (", getsize(SPrime), " prime numbers.)")

end-model
```

This example uses a new function, `getsize`, that if applied to a set returns the number of elements of the set. The condition in the `while` loop is the logical negation of an expression, marked with `not`: the loop is repeated as long as the condition `n in SNumbers` is not satisfied.

8.2.1 Set operators

The preceding example introduces the operator += to add sets to a set (there is also an operator -= to remove subsets from a set). Another set operator used in the example is `in` denoting that a single object is contained in a set. We have already encountered this operator in the enumeration of indices for the `forall` loop.

Mosel also defines the standard operators for comparing sets: subset (<=), superset (>=), difference (<>), end equality (=). Their use is illustrated by the following example:

```
model "Set comparisons"

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

 writeln("BRIGHT is included in RAINBOW: ", BRIGHT <= RAINBOW)
 writeln("RAINBOW is a superset of DARK: ", RAINBOW >= DARK)
 writeln("BRIGHT is different from DARK: ", BRIGHT <> DARK)
 writeln("BRIGHT is the same as RAINBOW: ", BRIGHT = RAINBOW)

end-model
```

As one might have expected, this example produces the following output:

```
BRIGHT is included in RAINBOW: true
```

```
RAINBOW is a superset of DARK: false
BRIGHT is different from DARK: true
BRIGHT is the same as RAINBOW: false
```

Chapter 9

Functions and procedures

When programs grow larger than the small examples presented so far, it becomes necessary to introduce some structure that makes them easier to read and to maintain. Usually, this is done by dividing the tasks that have to be executed into subtasks which may again be subdivided, and indicating the order in which these subtasks have to be executed and which are their activation conditions. To facilitate this structured approach, Mosel provides the concept of **subroutines**. Using subroutines, longer and more complex programs can be broken down into smaller subtasks that are easier to understand and to work with. Subroutines may be employed in the form of procedures or functions. **Procedures** are called as a program statement, they have no return value, **functions** must be called in an expression that uses their return value.

Mosel provides a set of predefined subroutines (for a comprehensive documentation the reader is referred to the Mosel Reference Manual), and it is possible to define new functions and procedures according to the needs of a specific program. A procedure that has occured repeatedly in this document is writeln. Typical examples of functions are mathematical functions like `abs`, `floor`, `ln`, `sin` *etc.*

9.1 Subroutine definition

User defined subroutines in Mosel have to be marked with `procedure`/`end-procedure` and `function`/`end-function` respectively. The return value of a function has to be assigned to `returned` as shown in the following example.

```
model "Simple subroutines"

 declarations
  a:integer
 end-declarations

 function three:integer
  returned := 3
 end-function

 procedure print_start
  writeln("The program starts here.")
 end-procedure

 print_start
 a:=three
 writeln("a = ", a)

end-model
```

This program will produce the following output:

```
The program starts here.
a = 3
```

9.2 Parameters

In many cases, the actions to be performed by a procedure or the return value expected from a function depend on the current value of one or several objects in the calling program. It is therefore possible to pass parameters into a subroutine. The (list of) parameter(s) is added in parantheses behind the name of the subroutine:

```
function times_two(b:integer):integer
 returned := 2*b
end-function
```

The structure of subroutines being very similar to the one of `model`, they may also include `declarations` sections for declaring **local parameters** that are only valid in the corresponding subroutine. It should be noted that such local parameters may **mask** global parameters within the scope of a subroutine, but they have no effect on the definition of the global parameter outside of the subroutine as is shown below in the extension of the example 'Simple subroutines'. Whilst it is not possible to modify function/procedure parameters in the corresponding subroutine, as in other programming languages the declaration of local parameters may **hide** these parameters. Mosel considers this as a possible mistake and prints a warning during compilation (without any consequence for the execution of the program).

```
model "Simple subroutines"

 declarations
  a:integer
 end-declarations

 function three:integer
  returned := 3
 end-function

 function times_two(b:integer):integer
  returned := 2*b
 end-function

 procedure print_start
  writeln("The program starts here.")
 end-procedure

 procedure hide_a_1
  declarations
   a: integer
  end-declarations

  a:=7
  writeln("Procedure hide_a_1: a = ", a)
 end-procedure

 procedure hide_a_2(a:integer)
  writeln("Procedure hide_a_2: a = ", a)
 end-procedure

 procedure hide_a_3(a:integer)
  declarations
   a: integer
  end-declarations

  a := 15
  writeln("Procedure hide_a_3: a = ", a)
 end-procedure

 print_start
```

```
 a:=three
 writeln("a = ", a)
 a:=times_two(a)
 writeln("a = ", a)
 hide_a_1
 writeln("a = ", a)
 hide_a_2(-10)
 writeln("a = ", a)
 hide_a_3(a)
 writeln("a = ", a)

end-model
```

During the compilation we get the warning

```
Mosel: W-165 at (30,3) of ‘subrout.mos’: Declaration of ‘a’ hides a parameter.
```

This is due to the redefinition of the parameter in procedure `hide_a_3`. The program results in the following output:

```
The program starts here.
a = 3
a = 6
Procedure hide_a_1: a = 7
a = 6
Procedure hide_a_2: a = -10
a = 6
Procedure hide_a_3: a = 15
a = 6
```

9.3 Recursion

The following example returns the largest common divisor of two numbers, just like the example ‘Lcdiv1’ in the previous chapter. This time we implement this task using recursive function calls, that is, from within function `lcdiv` we call again function `lcdiv`.

```
model Lcdiv2

 function lcdiv(A,B:integer):integer
  if(A=B) then
   returned:=A
  elif(A>B) then
   returned:=lcdiv(B,A-B)
  else
   returned:=lcdiv(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("Largest common divisor: ", lcdiv(A,B))

end-model
```

This example uses a simple recursion (a subroutine calling itself). In Mosel, it is also possible to use **cross-recursion**, that is, subroutine A calls subroutine B which again calls A. The only pre-requisite is that any subroutine that is called prior to its definition must be declared before it is called by using the `forward` statement (see below).

9.4 forward

A subroutine has to be 'known' at the place where it is called in a program. In the preceding examples we have defined all subroutines at the start of the programs but this may not always be feasible or desirable. Mosel therefore enables the user to declare a subroutine seperately from its definition by using the keyword `forward`. The **declaration** of 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** that must follow later in the program contains the body of the subroutine, that is, the actions to be executed by the subroutine.

The following example implements a **quick sort** algorithm for sorting a randomly generated array of numbers into ascending order. The procedure `qsort` that starts the sorting algorithm is defined at the very end of the program, it therefore needs to be declared at the beginning, before it is called. Procedure `qsort_start` calls the main sorting routine, `qsort`. Since the definition of this procedure precedes the place where it is called there is no need to declare it (but it still could be done). Procedure qsort calls yet again another subroutine, swap.

The idea of the quick sort algorithm is to partition the array that is to be sorted into two parts. The 'left' part containing all values smaller than the partitioning value and the 'right' part all the values that are larger than this value. The partitioning is then applied to the two subarrays, and so on, until all values are sorted.

```
model "Quick sort 1"

 parameters
  LIM=50
 end-parameters

 forward procedure qsort_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)
 qsort_start(T)
 writeln(T)

! 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)
  v:=L((s+e) div 2)             ! Determine the partitioning value
  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 qsort_start(L:array(r:range) of integer)
  qsort(L,getfirst(r),getlast(r))
 end-procedure

end-model
```

The quick sort example above demonstrates typical uses of subroutines, namely grouping actions that are executed repeatedly (qsort) and isolating subtasks (swap) in order to structure a program and increase its readability.

The calls to the procedures in this example are nested (procedure swap is called from qsort which is called from qsort_start): in Mosel there is no limit as to the number of nested calls to subroutines (it is not possible, though, to define subroutines within a subroutine).

9.5 Overloading of subroutines

In Mosel, it is possible to re-use the names of subroutines, provided that every version has a different number and/or types of parameters. This functionality is commonly referred to as **overloading**.

An example of an overloaded function in Mosel is getsol: if a variable is passed as a parameter it returns its solution value, if the parameter is a constraint the function returns the evaluation of the corresponding linear expression using the current solution.

Function abs (for obtaining the absolute value of a number) has different return types depending on the type of the input parameter: if an integer is input it returns an integer value, if it is called with a real value as input parameter it returns a real.

Function getcoeff is an example of a function that takes different numbers of parameters: if called with a single parameter (of type linctr) it returns the constant term of the input constraint, if a constraint and a variable are passed as parameters it returns the coefficient of the variable in the given constraint.

The user may define (additional) overloaded versions of any subroutines defined by Mosel as well as for his own functions and procedures. Note that it is not possible to overload a function with a procedure and *vice versa*.

Using the possibility to overload subroutines, we may rewrite the preceding example 'Quick sort' as follows.

```
model "Quick sort 2"

 parameters
  LIM=50
 end-parameters

 forward procedure qsort(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)
 qsort(T)
 writeln(T)
```

```
 procedure swap(L:array(range) of integer,i,j:integer)
  (...)                        (same procedure body as in the preceding example)
 end-procedure

 procedure qsort(L:array(range) of integer,s,e:integer)
  (...)                        (same procedure body as in the preceding example)
 end-procedure

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

end-model
```

The procedure `qsort_start` is now also called `qsort`. The procedure bearing this name in the first implementation keeps its name too; it has got two additional parameters which suffice to ensure that the right version of the procedure is called. To the contrary, it is not possible to give procedure swap the same name `qsort` because it takes exactly the same parameters as the original procedure `qsort` and hence it would not be possible to differentiate between these two procedures any more.

Chapter 10

Output

10.1 Producing formatted output

In some of the previous examples the procedures `write` and `writeln` have been used for displaying data, solution values and some accompanying text. To produce better formatted output, these procedures can be combined with the formatting procedure `strfmt`. In its simplest form, `strfmt`'s second argument indicates the (minimum) space reserved for writing the first argument and its placement within this space (negative values mean left justified printing, positive right justified). When writing a `real`, a third argument may be used to specify the maximum number of digits after the decimal point.

For example, if file `fo.mos` contains

```
model FO
 parameters
  r = 1.0         ! A real
  i = 0           ! An integer
 end-parameters

 writeln("i is ", i)
 writeln("i is ", strfmt(i,6) )
 writeln("i is ", strfmt(i,-6) )
 writeln("r is ", r)
 writeln("r is ", strfmt(r,6) )
 writeln("r is ",strfmt(r,10,4) )
end-model
```

and we run Mosel thus:

```
mosel -s -c "exec fo 'i=123, r=1.234567'"
```

we get output

```
i is 123
i is    123
i is 123
r is 1.23457
r is 1.23457
r is     1.2346
```

The following example prints out the solution of model 'Transport' (Section 3.1) in table format. The reader may be reminded that the objective of this problem is to compute the product flows from a set of plants (`PLANT`) to a set of sales regions (`REGION`) so as to minimize the total cost. The solution needs to comply with the capacity limits of the plants (`PLANTCAP`) and satisfy the demand `DEMAND` of all regions.

```
procedure print_table

 declarations
  rsum: array(REGION) of integer    ! Auxiliary data table for printing
  psum,prsum,ct,iflow: integer      ! Counters
 end-declarations

        ! Print heading and the first line of the table
 writeln("\nProduct Distribution\n--------------------")
 writeln(strfmt("Sales Region",44))
 write(strfmt("",14))
 forall(r in REGION) write(strfmt(r,9))
 writeln(strfmt("TOTAL",9), " Capacity")

        ! Print the solution values of the flow variables and
        ! calculate totals per region and per plant
 ct:=0
 forall(p in PLANT) do
   ct += 1
   if ct=2 then
     write("Plant ",strfmt(p,-8))
   else
     write("      ",strfmt(p,-8))
   end-if
   psum:=0
   forall(r in REGION) do
     iflow:=integer(getsol(flow(p,r)))
     psum += iflow
     rsum(r) += iflow
     if iflow<>0 then
       write(strfmt(iflow,9))
     else
       write("         ")
     end-if
   end-do
   writeln(strfmt(psum,9), strfmt(integer(PLANTCAP(p)),9))
 end-do

        ! Print the column totals
 write("\n", strfmt("TOTAL",-14))
 prsum:=0
 forall(r in REGION) do
   prsum += rsum(r);
   write(strfmt(rsum(r),9))
 end-do
 writeln(strfmt(prsum,9))

        ! Print demand of every region
 write(strfmt("Demand",-14))
 forall(r in REGION) write(strfmt(integer(DEMAND(r)),9))

        ! Print objective function value
 writeln("\n\nTotal cost of distribution = ", strfmt(getobjval/1e6,0,3),
         " million.")

end-procedure
```

With the data from Chapter 3 the procedure `print_table` produces the following output:

```
Product Distribution
--------------------
                                Sales Region
              Scotland    North    SWest    SEast Midlands    TOTAL Capacity
      Corby                          180      820     2000     3000     3000
Plant Deeside                1530    920               250     2700     2700
      Glasgow     2840       1270                              4110     4500
      Oxford                         1500     2000     500     4000     4000

TOTAL             2840       2800    2600     2820    2750    13810
Demand            2840       2800    2600     2820    2750

Total cost of distribution = 81.018 million.
```

10.2 File output

If we do not want the output of procedure `print_tab` in the previous section to be displayed on screen but to be saved in the file `out.txt`, we simply open the file for writing at the beginning of the procedure by adding

```
fopen("out.txt",F_OUTPUT)
```

before the first `writeln` statement, and close it at the end of the procedure, after the last `writeln` statement with

```
fclose(F_OUTPUT)
```

If we do not want any existing contents of the file `out.txt` to be deleted, so that the result table is appended to the end of the file, we need to write the following for opening the file (closing it the same way as before):

```
fopen("out.txt",F_OUTPUT+F_APPEND)
```

As with input of data from file, there are several ways of outputting data (*e.g.* solution values) to a file in Mosel. The following example demonstrates three different ways of writing the contents of an array A to a file.

```
model "Trio output"
 uses "mmetc"

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

 A := [ 2,  4,  6,
       12, 14, 16,
       22, 24, 26]

! First method: use an initializations block
 initializations to "out_1.dat"
  A as "MYOUT"
 end-initializations

! Second method: use the built-in writeln function
 fopen("out_2.dat",F_OUTPUT)
 forall(i,j in 1..3)
  writeln('A_out(',i,' and ',j,') = ', A(i,j))
 fclose(F_OUTPUT)

! Third method: use diskdata
 diskdata(ETC_OUT+ETC_SPARSE,"out_3.dat", A)

end-model
```

File out_1.dat will contain the following:

```
'MYOUT': [2 4 6 12 14 16 22 24 26]
```

If this file contains already a data entry myout, it is replaced with this output without modifying or deleting any other contents of this file. Otherwise, the output is appended at the end of it.

The nicely formatted output to out_2.dat results in the following:

```
A_out(1 and 1) = 2
A_out(1 and 2) = 4
A_out(1 and 3) = 6
A_out(2 and 1) = 12
A_out(2 and 2) = 14
A_out(2 and 3) = 16
A_out(3 and 1) = 22
A_out(3 and 2) = 24
A_out(3 and 3) = 26
```

The output with diskdata simply prints the contents of the array to out_3.dat, with option ETC_SPARSE each entry is preceded by the corresponding indices:

```
1,1,2
1,2,4
1,3,6
2,1,12
2,2,14
2,3,16
3,1,22
3,2,24
3,3,26
```

Without option ETC_SPARSE out_3.dat looks as follows:

```
2,4,6
12,14,16
22,24,26
```

Chapter 11

More about Integer Programming

This chapter presents two applications to (Mixed) Integer Programming of the programming facilities in Mosel that have been introduced in the previous chapters.

11.1 Cut generation

Cutting plane methods add constraints (cuts) to the problem that cut off parts of the convex hull of the integer solutions, thus drawing the solution of the LP relaxation closer to the integer feasible solutions and improving the bound provided by the solution of the relaxed problem.

The Xpress-Optimizer provides automated cut generation (see the optimizer documentation for details). To show the effects of the cuts that are generated by our example we switch off the automated cut generation.

11.1.1 Example problem

The problem we want to solve is the following: a large company is planning to outsource the cleaning of its offices at the least cost. The *NSITES* office sites of the company are grouped into areas (set $AREAS = \{1, \ldots, NAREAS\}$). Several professional cleaning companies (set $CONTR = \{1, \ldots, NCONTRACTORS\}$) have submitted bids for the different sites, a cost of 0 in the data meaning that a contractor is not bidding for a site.

To avoid being dependent on a single contractor, adjacent areas have to be allocated to different contractors. Every site s (s in $SITES = \{1, \ldots, NSITES\}$) is to be allocated to a single contractor, but there may be between $LOWCON_a$ and $UPPCON_a$ contractors per area a.

11.1.2 Model formulation

For the mathematical formulation of the problem we introduce two sets of variables:

$clean_{cs}$ indicates whether contractor c is cleaning site s
$alloc_{ca}$ indicates whether contractor c is allocated any site in area a

The objective to minimize the total cost of all contracts is as follows (where $PRICE_{sc}$ is the price per site and contractor):

$$\text{minimize} \sum_{c \in CONTR} \sum_{s \in SITES} PRICE_{sc} \cdot clean_{cs}$$

We need the following three sets of constraints to formulate the problem:

1. Each site must be cleaned by exactly one contractor.

$$\forall s \in SITES : \sum_{c \in CONTR} clean_{cs} = 1$$

2. Adjacent areas must not be allocated to the same contractor.

$$\forall c \in CONTR, a, b \in AREAS, a > b \text{ and } ADJACENT(a, b) = 1 : \quad alloc_{ca} + alloc_{cb} \leq 1$$

3. The lower and upper limits on the number of contractors per area must be respected.

$$\forall a \in AREAS : \sum_{c \in CONTR} alloc_{ca} \geq LOWCON_a$$
$$\forall a \in AREAS : \sum_{c \in CONTR} alloc_{ca} \leq UPPCON_a$$

To express the relation between the two sets of variables we need more constraints: a contractor c is allocated to an area a if and only if he is allocated a site s in this area, that is, y_{ca} is 1 if and only if some x_{cs} (for a site s in area a) is 1. This equivalence is expressed by the following two sets of constraints, one for each sense of the implication ($AREA_s$ is the area a site s belongs to and $NUMSITE_a$ the number of sites in area a):

$$\forall c \in CONTR, a \in AREAS : alloc_{ca} \leq \sum_{\substack{s \in SITES \\ AREA_s = a}} clean_{cs}$$
$$\forall c \in CONTR, a \in AREAS : alloc_{ca} \geq \frac{1}{NUMSITE_a} \cdot \sum_{\substack{s \in SITES \\ AREA_s = a}} clean_{cs}$$

11.1.3 Implementation

The resulting Mosel program is the following. The variables $clean_{cs}$ are defined as a **dynamic array** and are only created if contractor c bids for site s (that is, $PRICE_{sc} > 0$ or, taking into account inaccuracies in the data, $PRICE_{sc} > 0.01$).

Another implementation detail that the reader may notice is the separate initialization of the array sizes: we are thus able to create all arrays with fixed sizes (with the exception of the previously mentioned array of variables that is explicitly declared dynamic). This allows Mosel to handle them in a more efficient way.

```
model "Office cleaning"
 uses "mmxprs","mmsystem"

 declarations
  PARAM: array(1..3) of integer
 end-declarations

 initializations from 'clparam.dat'
  PARAM
 end-initializations

 declarations
  NSITES = PARAM(1)                    ! Number of sites
  NAREAS = PARAM(2)                    ! Number of areas (subsets of sites)
  NCONTRACTORS = PARAM(3)              ! Number of contractors
  AREAS = 1..NAREAS
  CONTR = 1..NCONTRACTORS
  SITES = 1..NSITES
  AREA: array(SITES) of integer        ! Area site is in
  NUMSITE: array(AREAS) of integer     ! Number of sites in an area
  LOWCON: array(AREAS) of integer      ! Lower limit on the number of
                                       ! contractors per area
  UPPCON: array(AREAS) of integer      ! Upper limit on the number of
                                       ! contractors per area
  ADJACENT: array(AREAS,AREAS) of integer    ! 1 if areas adjacent, 0 otherwise
  PRICE: array(SITES,CONTR) of real    ! Price per contractor per site
```

```
  clean: dynamic array(CONTR,SITES) of mpvar ! 1 iff contractor c cleans site s
  alloc: array(CONTR,AREAS) of mpvar    ! 1 iff contractor allocated to a site
                                        !  in area a
 end-declarations

 initializations from 'cldata.dat'
  [NUMSITE,LOWCON,UPPCON] as 'AREA'
  ADJACENT
  PRICE
 end-initializations

 ct:=1
 forall(a in AREAS) do
  forall(s in ct..ct+NUMSITE(a)-1)
   AREA(s):=a
  ct+= NUMSITE(a)
 end-do

 forall(c in CONTR, s in SITES | PRICE(s,c) > 0.01) create(clean(c,s))

! Objective: Minimize total cost of all cleaning contracts
 Cost:= sum(c in CONTR, s in SITES) PRICE(s,c)*clean(c,s)

! Each site must be cleaned by exactly one contractor
 forall(s in SITES) sum(c in CONTR) clean(c,s) = 1

! Ban same contractor from serving adjacent areas
 forall(c in CONTR, a,b in AREAS | a > b and ADJACENT(a,b) = 1)
  alloc(c,a) + alloc(c,b) <= 1

! Specify lower & upper limits on contracts per area
 forall(a in AREAS | LOWCON(a)>0)
  sum(c in CONTR) alloc(c,a) >= LOWCON(a)
 forall(a in AREAS | UPPCON(a)<NCONTRACTORS)
  sum(c in CONTR) alloc(c,a) <= UPPCON(a)

! Define alloc(c,a) to be 1 iff some clean(c,s)=1 for sites s in area a
 forall(c in CONTR, a in AREAS) do
  alloc(c,a) <= sum(s in SITES| AREA(s)=a) clean(c,s)
  alloc(c,a) >= 1.0/NUMSITE(a) * sum(s in SITES| AREA(s)=a) clean(c,s)
 end-do

 forall(c in CONTR) do
  forall(s in SITES) clean(c,s) is_binary
  forall(a in AREAS) alloc(c,a) is_binary
 end-do

 minimize(Cost)                 ! Solve the MIP problem
end-model
```

In the preceding model, we have chosen to implement the constraints that force the variables $alloc_{ca}$ to become 1 whenever a variable $clean_{cs}$ is 1 for some site s in area a in an aggregated way (this type of constraint is usually referred to as Multiple Variable Lower Bound, MVLB, constraints). Instead of

```
  forall(c in CONTR, a in AREAS)
   alloc(c,a) >= 1.0/NUMSITE(a) * sum(s in SITES| AREA(s)=a) clean(c,s)
```

we could also have used the stronger formulation

```
  forall(c in CONTR, s in SITES)
   alloc(c,AREA(s)) >= clean(c,s)
```

but this considerably increases the total number of constraints. The aggregated constraints are sufficient to express this problem, but this formulation is very loose, with the consequence that the solution of the LP relaxation only provides a very bad approximation of the integer solution that we want to obtain. For large data sets the Branch-and-Bound search may therefore take a long time.

11.1.4 Cut-and-Branch

To improve this situation without blindly adding many unnecessary constraints, we implement a cut generation loop at the top node of the search that only adds those constraints that are violated be the current LP solution.

The cut generation loop (procedure `top_cut_gen`) performs the following steps:

- solve the LP and save the basis
- get the solution values
- identify violated constraints and add them to the problem
- load the modified problem and load the previous basis

```
 procedure top_cut_gen
  declarations
   MAXCUTS = 2500              ! Max no. of constraints added in total
   MAXPCUTS = 1000             ! Max no. of constraints added per pass
   MAXPASS  = 50               ! Max no. passes
   ncut, npass, npcut: integer        ! Counters for cuts and passes
   feastol: real                      ! Zero tolerance
   solc: array(CONTR,SITES) of real   ! Sol. values for variables ‘clean’
   sola: array(CONTR,AREAS) of real   ! Sol. values for variables ‘alloc’
   objval,starttime: real
   cut: array(range) of linctr
  end-declarations

  starttime:=gettime
  setparam("XPRS_CUTSTRATEGY", 0)     ! Disable automatic cuts
  setparam("XPRS_PRESOLVE", 0)        ! Switch presolve off
  feastol:= getparam("XPRS_FEASTOL")  ! Get the Optimizer zero tolerance
  setparam("ZEROTOL", feastol * 10)   ! Set the comparison tolerance of Mosel
  ncut:=0
  npass:=0

  while (ncut<MAXCUTS and npass<MAXPASS) do
    npass+=1
    npcut:= 0
    minimize(XPRS_LIN, Cost)          ! Solve the LP
    if (npass>1 and objval=getobjval) then break; end-if
    savebasis(1)                      ! Save the current basis
    objval:= getobjval                ! Get the objective value

    forall(c in CONTR) do             ! Get the solution values
      forall(a in AREAS) sola(c,a):=getsol(alloc(c,a))
      forall(s in SITES) solc(c,s):=getsol(clean(c,s))
    end-do

! Search for violated constraints and add them to the problem:
    forall(c in CONTR, s in SITES)
     if solc(c,s) > sola(c,AREA(s)) then
      cut(ncut):= alloc(c,AREA(s)) >= clean(c,s)
      ncut+=1
      npcut+=1
      if (npcut>MAXPCUTS or ncut>MAXCUTS) then break 2; end-if
     end-if
```

```
    writeln("Pass ", npass, " (", gettime-starttime, " sec), objective value ",
            objval, ", cuts added: ", npcut, " (total ", ncut,")")

    if npcut=0 then
      break
    else
      loadprob(Cost)                    ! Reload the problem
      loadbasis(1)                      ! Load the saved basis
    end-if
  end-do

                                         ! Display cut generation status
  write("Cut phase completed: ")
  if (ncut >= MAXCUTS) then writeln("space for cuts exhausted")
  elif (npass >= MAXPASS) then writeln("maximum number of passes reached")
  else writeln("no more violations or no improvement to objective")
  end-if
 end-procedure
```

Assuming we add the definition of procedure `top_cut_gen` to the end of our model, we need to add its declaration at the beginning of the model:

```
forward procedure topcutgen
```

and the call to this function immediately before the optimization:

```
top_cut_gen                         ! Constraint generation at top node
minimize(Cost)                      ! Solve the MIP problem
```

Since we wish to use our own cut strategy, we switch off the default cut generation in Xpress-Optimizer:

```
setparam("XPRS_CUTSTRATEGY", 0)
```

We also turn the presolve off since we wish to access the solution to the original problem after solving the LP-relaxations:

```
setparam("XPRS_PRESOLVE", 0)
```

11.1.5 Comparison tolerance

In addition to the parameter settings we also retrieve the feasibility tolerance used by Xpress-Optimizer: the Optimizer works with tolerance values for integer feasibility and solution feasibility that are typically of the order of 10^{-6} by default. When evaluating a solution, for instance by performing comparisons, it is important to take into account these tolerances.

After retrieving the feasibility tolerance of the Optimizer we set the comparison tolerance of Mosel (`ZEROTOL`) to this value. This means, for example, the test $x = 0$ evaluates to true if x lies between -*ZEROTOL* and *ZEROTOL*, $x \leq 0$ is true if the value of x is at most *ZEROTOL*, and $x > 0$ is fulfilled if x is greater than *ZEROTOL*.

Comparisons in Mosel always use a tolerance, with a very small default value. By resetting this parameter to the Optimizer feasibility tolerance Mosel evaluates solution values just like the Optimizer.

11.1.6 Branch-and-Cut

The cut generation loop presented in the previous subsection only generates violated inequalities at the top node before entering the Branch-and-Bound search and adds them to the problem in the form of additional constraints. We may do the same using the **cut manager** of Xpress-Optimizer. In this case, the violated constraints are added to the problem via the **cut pool**. We may even generate and add cuts during the Branch-and-Bound search. A cut added at a node using `addcuts` only applies to this node and its descendants, so one may use this functionality to define **local cuts** (however, in our example, all generated cuts are valid globally).

The cut manager is set up with a call to procedure `tree_cut_gen` before starting the optimization (preceded by the declaration of the procedure using `forward` earlier in the program). To avoid initializing the solution arrays and the feasibility tolerance repeatedly, we now turn these into globally defined objects:

```
declarations
 feastol: real                         ! Zero tolerance
 solc: array(CONTR,SITES) of real      ! Sol. values for variables 'clean'
 sola: array(CONTR,AREAS) of real      ! Sol. values for variables 'alloc'
end-declarations

tree_cut_gen                           ! Set up cut generation in B&B tree
minimize(Cost)                         ! Solve the MIP problem
```

As we have seen before, procedure `tree_cut_gen` disables the default cut generation and turns presolve off. It also indicates the number of extra rows to be reserved in the matrix for the cuts we are generating:

```
procedure tree_cut_gen
 setparam("XPRS_CUTSTRATEGY", 0)     ! Disable automatic cuts
 setparam("XPRS_PRESOLVE", 0)        ! Switch presolve off
 setparam("XPRS_EXTRAROWS", 5000)    ! Reserve extra rows in matrix

 feastol:= getparam("XPRS_FEASTOL")  ! Get the zero tolerance
 setparam("zerotol", feastol * 10)   ! Set the comparison tolerance of Mosel

 setcallback(XPRS_CB_CM, "cb_node")
end-procedure
```

The last line of this procedure defines the **cut manager entry callback** function that will be called by the optimizer from every node of the Branch-and-Bound search tree. This cut generation routine (function cb_node) performs the following steps:

- get the solution values
- identify violated inequalities and add them to the problem

It is implemented as follows (we restrict the generation of cuts to the first three levels, *i.e.* `depth < 4`, of the search tree):

```
 function cb_node:boolean

  declarations
   ncut: integer                     ! Counters for cuts
   cut: array(range) of linctr       ! Cuts
   cutid: array(range) of integer    ! Cut type identification
   type: array(range) of integer     ! Cut constraint type
  end-declarations

  returned:=false                    ! Call this function once per node

  depth:=getparam("XPRS_NODEDEPTH")
  node:=getparam("XPRS_NODES")

  if depth<4 then
   ncut:=0

! Get the solution values
   setparam("XPRS_SOLUTIONFILE",0)
   forall(c in CONTR) do
```

```
    forall(a in AREAS) sola(c,a):=getsol(alloc(c,a))
    forall(s in SITES) solc(c,s):=getsol(clean(c,s))
   end-do
   setparam("XPRS_SOLUTIONFILE",1)

! Search for violated constraints
   forall(c in CONTR, s in SITES)
    if solc(c,s) > sola(c,AREA(s)) then
     cut(ncut):= alloc(c,AREA(s)) - clean(c,s)
     cutid(ncut):= 1
     type(ncut):= CT_GEQ
     ncut+=1
    end-if

! Add cuts to the problem
   if ncut>0 then
    returned:=true                     ! Call this function again
    addcuts(cutid, type, cut);
    writeln("Cuts added : ", ncut, " (depth ", depth, ", node ", node,
           ", obj. ", getparam("XPRS_LPOBJVAL"), ")")
   end-if
  end-if

 end-function
```

The prototype of this function is prescribed by the type of the callback (see the Xpress-Optimizer Reference Manual and the chapter on `mmxprs` in the Mosel Language Reference Manual). At every node this function is called repeatedly, followed by a re-solution of the current LP, as long as it returns `true`.

Remark: if one wishes to access the solution values in a callback function, the Xpress-Optimizer parameter `XPRS_SOLUTIONFILE` must be set to 0 before getting the solution and after getting the solutions it must be set back to 1.

11.2 Column generation

The technique of column generation is used for solving linear problems with a huge number of variables for which it is not possible to generate explicitly all columns of the problem matrix. Starting with a very restricted set of columns, after each solution of the problem a column generation algorithm adds one or several columns that improve the current solution. These columns must have a negative reduced cost (in a minimization problem) and are calculated based on the dual value of the current solution.

For solving large MIP problems, column generation typically has to be combined with a Branch-and-Bound search, leading to a so-called Branch-and-Price algorithm. The example problem described below is solved by solving a sequence of LPs without starting a tree search.

11.2.1 Example problem

A paper mill produces rolls of paper of a fixed width *MAXWIDTH* that are subsequently cut into smaller rolls according to the customer orders. The rolls can be cut into *NWIDTHS* different sizes. The orders are given as demands for each width i ($DEMAND_i$). The objective of the paper mill is to satisfy the demand with the smallest possible number of paper rolls in order to minimize the losses.

11.2.2 Model formulation

The objective of minimizing the total number of rolls can be expressed as choosing the best set of cutting patterns for the current set of demands. Since it may not be obvious how to calculate all possible cutting patterns by hand, we start off with a basic set of patterns ($PATTERNS_1$,..., $PATTERNS_{NWIDTH}$), that consists of cutting small rolls all of the same width as many times as possible out of the large roll. This type of problem is called a **cutting stock problem**.

If we define variables use_j to denote the number of times a cutting pattern j ($j \in WIDTHS = \{1, \ldots, NWIDTH\}$) is used, then the objective becomes to minimize the sum of these variables, subject to the constraints that the demand for every size has to be met.

$$\text{minimize} \sum_{j \in WIDTHS} use_j$$

$$\sum_{j \in WIDTHS} PATTERNS_{ij} \cdot use_j \geq DEMAND_i$$

$$\forall j \in WIDTHS : use_j \leq \text{ceil}(DEMAND_j \;/\; PATTERNS_{jj}), \quad use_j \text{ integer}$$

Function *ceil* means rounding to the next larger integer value.

11.2.3 Implementation

The first part of the Mosel model implementing this problem looks as follows:

```
model Papermill
 uses "mmxprs"

 forward procedure column_gen
 forward function knapsack(C:array(range) of real, A:array(range) of real,
                          B:real, xbest:array(range) of integer,
                          pass: integer): real
 forward procedure show_new_pat(dj:real, vx: array(range) of integer)

 declarations
  NWIDTHS = 5                          ! Number of different widths
  WIDTHS = 1..NWIDTHS                  ! Range of widths
  RP: range                            ! Range of cutting patterns
  MAXWIDTH = 94                        ! Maximum roll width
  EPS = 1e-6                           ! Zero tolerance

  WIDTH: array(WIDTHS) of real         ! Possible widths
  DEMAND: array(WIDTHS) of integer     ! Demand per width
  PATTERNS: array(WIDTHS,WIDTHS) of integer ! (Basic) cutting patterns

  use: array(RP) of mpvar              ! Rolls per pattern
  soluse: array(RP) of real            ! Solution values for variables 'use'
  Dem: array(WIDTHS) of linctr         ! Demand constraints
  MinRolls: linctr                     ! Objective function

  KnapCtr, KnapObj: linctr             ! Knapsack constraint+objective
  x: array(WIDTHS) of mpvar            ! Knapsack variables
 end-declarations

 WIDTH:=  [ 17, 21, 22.5,  24, 29.5]
 DEMAND:= [150, 96,   48, 108,  227]
                                       ! Make basic patterns
 forall(j in WIDTHS)  PATTERNS(j,j) := floor(MAXWIDTH/WIDTH(j))

 forall(j in WIDTHS) do
  create(use(j))                       ! Create NWIDTHS variables 'use'
  use(j) is_integer                    ! Variables are integer and bounded
  use(j) <= integer(ceil(DEMAND(j)/PATTERNS(j,j)))
 end-do

 MinRolls:= sum(j in WIDTHS) use(j)     ! Objective: minimize no. of rolls
```

```
forall(i in WIDTHS)                       ! Satisfy all demands
 Dem(i):= sum(j in WIDTHS) PATTERNS(i,j) * use(j) >= DEMAND(i)

column_gen                                ! Column generation at top node

minimize(MinRolls)                        ! Compute the best integer solution
                                          ! for the current problem (including
                                          ! the new columns)
writeln("Best integer solution: ", getobjval, " rolls")
write("  Rolls per pattern: ")
forall(i in RP) write(getsol(use(i)),", ")
```

The paper mill can satisfy the demand with just the basic set of cutting patterns, but it is likely to incur significant losses through wasting more than necessary of every large roll and by cutting more small rolls than its customers have ordered. We therefore employ a column generation heuristic to find more suitable cutting patterns.

The following procedure `column_gen` defines a column generation loop that is executed at the top node (this heuristic was suggested by M. Savelsbergh for solving a similar cutting stock problem). The column generation loop performs the following steps:

- solve the LP and save the basis
- get the solution values
- compute a more profitable cutting pattern based on the current solution
- generate a new column (= cutting pattern): add a term to the objective function and to the corresponding demand constraints
- load the modified problem and load the saved basis

To be able to increase the number of variables use_j in this function, these variables have been declared at the beginning of the program as a **dynamic array** without specifying any index range.

By setting Mosel's comparison tolerance to *EPS*, the test $zbest = 0$ checks whether *zbest* lies within *EPS* of 0 (see explanation in Section 11.1).

```
procedure column_gen
  declarations
   dualdem: array(WIDTHS) of real
   xbest: array(WIDTHS) of integer
   dw, zbest, objval: real
  end-declarations

  setparam("XPRS_CUTSTRATEGY", 0)         ! Disable automatic cuts
  setparam("XPRS_PRESOLVE", 0)            ! Switch presolve off
  setparam("zerotol", EPS)                ! Set comparison tolerance of Mosel
  npatt:=NWIDTHS
  npass:=1

  while(true) do
    minimize(XPRS_LIN, MinRolls)          ! Solve the LP

    savebasis(1)                          ! Save the current basis
    objval:= getobjval                    ! Get the objective value

                                          ! Get the solution values
    forall(j in 1..npatt)  soluse(j):=getsol(use(j))
    forall(i in WIDTHS)  dualdem(i):=getdual(Dem(i))
                                          ! Solve a knapsack problem
    zbest:= knapsack(dualdem, WIDTH, MAXWIDTH, xbest, npass) - 1.0
```

```
    write("Pass ", npass, ": ")
    if zbest = 0 then
      writeln("no profitable column found.\n")
      break
    else
      show_new_pat(zbest, xbest)         ! Print the new pattern
      npatt+=1
      create(use(npatt))                 ! Create a new var. for this pattern
      use(npatt) is_integer

      MinRolls+= use(npatt)              ! Add new var. to the objective
      dw:=0
      forall(i in WIDTHS)
        if xbest(i) > 0 then
         Dem(i)+= xbest(i)*use(npatt)    ! Add new var. to demand constr.s
         dw:= maxlist(dw, ceil(DEMAND(i)/xbest(i) ))
        end-if
      use(npatt) <= dw                   ! Set upper bound on the new var.

      loadprob(MinRolls)                 ! Reload the problem
      loadbasis(1)                       ! Load the saved basis
    end-if
    npass+=1
  end-do

  writeln("Solution after column generation: ", objval, " rolls, ",
          getsize(RP), " patterns")
  write("   Rolls per pattern: ")
  forall(i in RP) write(soluse(i),", ")
  writeln
 end-procedure
```

The preceding procedure `column_gen` calls the following auxiliary function `knapsack` to solve an **integer knapsack problem** of the form

$$\text{maximize } z = \sum_{j \in WIDTHS} C_i \cdot x_j$$

$$\sum_{j \in WIDTHS} A_j \cdot x_j \leq B,$$

$$\forall j \in WIDTHS : x_j \text{ integer}$$

The function `knapsack` solves a second optimization problem that is independent of the main, cutting stock problem since the two have no variables in common. We thus effectively work with **two** problems in a single Mosel model.

For efficiency reasons we have defined the knapsack variables and constraints globally. The integrality condition on the knapsack variables remains unchanged between several calls to this function, so we establish it when solving the first knapsack problem. On the other hand, the knapsack constraint and the objective function have different coefficients at every execution, so we need to replace them every time the function is called.

We **reset** the knapsack constraints to 0 at the end of this function so that they do not unnecessarily increase the size of the main problem. The same is true in the other sense: **hiding** the demand constraints while solving the knapsack problem makes life easier for the optimizer, but is not essential for getting the correct solution.

```
 function knapsack(C:array(range) of real, A:array(range) of real, B:real,
                   xbest:array(range) of integer, pass: integer):real

! Hide the demand constraints
  forall(j in WIDTHS) sethidden(Dem(j), true)
```

```
! Define the knapsack problem
  KnapCtr := sum(j in WIDTHS) A(j)*x(j) <= B
  KnapObj := sum(j in WIDTHS) C(j)*x(j)

! Integrality condition
  if(pass=1) then
   forall(j in WIDTHS) x(j) is_integer
  end-if

  maximize(KnapObj)
  returned:=getobjval
  forall(j in WIDTHS) xbest(j):=round(getsol(x(j)))

! Reset knapsack constraint and objective, unhide demand constraints
  KnapCtr := 0
  KnapObj := 0
  forall(j in WIDTHS) sethidden(Dem(j), false)
 end-function
```

To complete the model, we add the following procedure `show_new_pat` to print every new pattern we find.

```
 procedure show_new_pat(dj:real, vx: array(range) of integer)
  declarations
   dw: real
  end-declarations

  writeln("new pattern found with marginal cost ", dj)
  write("   Widths distribution: ")
  dw:=0
  forall(i in WIDTHS) do
    write(WIDTH(i), ":", vx(i), "  ")
    dw += WIDTH(i)*vx(i)
  end-do
  writeln("Total width: ", dw)
 end-procedure

end-model
```

Chapter 12

Extensions to Linear Programming

The two examples (recursion and Goal Programming) in this chapter show how Mosel can be used to implement extensions of Linear Programming.

12.1 Recursion

Recursion, more properly known as **Successive Linear Programming**, is a technique whereby LP may be used to solve certain non-linear problems. Some coefficients in an LP problem are defined to be functions of the optimal values of LP variables. When an LP problem has been solved, the coefficients are re-evaluated and the LP re-solved. Under some assumptions this process may converge to a local (though not necessarily a global) optimum.

12.1.1 Example problem

Consider the following financial planning problem: We wish to determine the yearly interest rate x so that for a given set of payments we obtain the final balance of 0. Interest is paid quarterly according to the following formula:

$$interest_t = (92 \;/\; 365) \cdot balance_t \cdot interest\_rate$$

The balance at time t ($t = 1, \ldots, T$) results from the balance of the previous period $t-1$ and the net of payments and interest:

$$\begin{aligned} net_t &= Payments_t - interest_t \\ balance_t &= balance_{t-1} - net_t \end{aligned}$$

12.1.2 Model formulation

This problem cannot be modeled just by LP because we have the T products

$$balance_t \cdot interest\_rate$$

which are non-linear. To express an approximation of the original problem by LP we replace the interest rate variable x by a (constant) guess X of its value and a deviation variable dx

$$x = X + dx$$

The formula for the quarterly interest payment i_t therefore becomes

$$\begin{aligned} interest_t &= 92 \;/\; 365 \cdot (balance_{t-1} \cdot x) \\ &= 92 \;/\; 365 \cdot (balance_{t-1} \cdot (X + dx)) \\ &= 92 \;/\; 365 \cdot (balance_{t-1} \cdot X + balance_{t-1} \cdot dx) \end{aligned}$$

where $balance_t$ is the balance at the beginning of period t.

We now also replace the balance $balance_{t-1}$ in the product with dx by a guess B_{t-1} and a deviation db_{t-1}

$$\begin{aligned} iinterest_t &= 92 / 365 \cdot (balance_{t-1} \cdot X + (B_{t-1} + db_{t-1}) \cdot dx) \\ &= 92 / 365 \cdot (balance_{t-1} \cdot X + B_{t-1} \cdot dx + db_{t-1} \cdot dx) \end{aligned}$$

which can be approximated by dropping the product of the deviation variables

$$interest_t = 92 / 365 \cdot (balance_{t-1} \cdot X + B_{t-1} \cdot dx)$$

To ensure feasibility we add penalty variables $eplus_t$ and $eminus_t$ for positive and negative deviations in the formulation of the constraint:

$$interest_t = 92 / 365 \cdot (balance_{t-1} \cdot X + B_{t-1} \cdot dx + eplus_t - eminus_t)$$

The objective of the problem is to get feasible, that is to minimize the deviations:

$$\text{minimize} \sum_{t \in QUARTERS} (eplus_t + eminus_t)$$

12.1.3 Implementation

The Mosel model then looks as follows (note the balance variables $balance_t$ as well as the deviation dx and the quarterly nets net_t are defined as free variables, that is, they may take any values between minus and plus infinity):

```
model Recurse
 uses "mmxprs"

 forward procedure solve_recurse

 declarations
  T=6                              ! Time horizon
  QUARTERS=1..T                    ! Range of time periods
  P,R,V: array(QUARTERS) of real   ! Payments
  B: array(QUARTERS) of real       ! Initial guess as to balances b(t)
  X: real                          ! Initial guess as to interest rate x

  interest: array(QUARTERS) of mpvar ! Interest
  net: array(QUARTERS) of mpvar      ! Net
  balance: array(QUARTERS) of mpvar  ! Balance
  x: mpvar                         ! Interest rate
  dx: mpvar                        ! Change to x
  eplus, eminus: array(QUARTERS) of mpvar  ! + and - deviations
 end-declarations

 X:= 0.00
 B:= [1, 1, 1, 1, 1, 1]
 P:= [-1000, 0, 0, 0, 0, 0]
 R:= [206.6, 206.6, 206.6, 206.6, 206.6, 0]
 V:= [-2.95, 0, 0, 0, 0, 0]
                                   ! net = payments - interest
 forall(t in QUARTERS) net(t) = (P(t)+R(t)+V(t)) - interest(t)

                                   ! Money balance across periods
 forall(t in QUARTERS) balance(t) = if(t>1, balance(t-1), 0) - net(t)

 forall(t in 2..T) Interest(t):=     ! Approximation of interest
   -(365/92)*interest(t) + X*balance(t-1) + B(t-1)*dx + eplus(t) - eminus(t) = 0
```

```
 Def:= X + dx = x                    ! Define the interest rate: x = X + dx

 Feas:= sum(t in QUARTERS) (eplus(t)+eminus(t))  ! Objective: get feasible

 interest(1) = 0                     ! Initial interest is zero
 forall (t in QUARTERS) net(t) is_free
 forall (t in 1..T-1) balance(t) is_free
 balance(T) = 0                      ! Final balance is zero
 dx is_free

 minimize(Feas)                      ! Solve the LP-problem

 solve_recurse                       ! Recursion loop

                                     ! Print the solution
 writeln("\nThe interest rate is ", getsol(x))
 write(strfmt("t",5), strfmt(" ",4))
 forall(t in QUARTERS) write(strfmt(t,5), strfmt(" ",3))
 write("\nBalances ")
 forall(t in QUARTERS)  write(strfmt(getsol(balance(t)),8,2))
 write("\nInterest ")
 forall(t in QUARTERS)  write(strfmt(getsol(interest(t)),8,2))

end-model
```

In the model above we have declared the procedure `solve_recurse` that executes the recursion but it has not yet been defined. The recursion on x and the $balance_t$ ($t = 1, \ldots, T-1$) is implemented by the following steps:

(a) The B_{t-1} in constraints $Interest_t$ get the prior solution value of $balance_{t-1}$
(b) The X in constraints $Interest_t$ get the prior solution value of x
(c) The X in constraint Def gets the prior solution value of x

We say we have **converged** when the change in dx (*variation*) is less than 0.000001 (*TOLERANCE*). By setting Mosel's comparison tolerance to this value the test $variation > 0$ checks whether *variation* is greater than *TOLERANCE*.

```
 procedure solve_recurse
  declarations
   TOLERANCE=0.000001                ! Convergence tolerance
   variation: real                   ! Variation of x
   BC: array(QUARTERS) of real
  end-declarations

  setparam("zerotol", TOLERANCE)     ! Set Mosel comparison tolerance
  variation:=1.0
  ct:=0

  while(variation>0) do
   savebasis(1)                      ! Save the current basis
   ct+=1
   forall(t in 2..T)
     BC(t-1):= getsol(balance(t-1))  ! Get solution values for balance(t)'s
   XC:= getsol(x)                    ! and x
   write("Round ", ct, " x:", getsol(x), " (variation:", variation,"), ")
   writeln("Simplex iterations: ", getparam("XPRS_SIMPLEXITER"))

   forall(t in 2..T) do              ! Update coefficients
     Interest(t)+= (BC(t-1)-B(t-1))*dx
     B(t-1):=BC(t-1)
     Interest(t)+= (XC-X)*balance(t-1)
   end-do
   Def+= XC-X
   X:=XC
   oldxval:=XC                       ! Store solution value of x
```

```
   loadprob(Feas)                     ! Reload the problem into the optimizer
   loadbasis(1)                       ! Reload previous basis
   minimize(Feas)                     ! Re-solve the LP-problem

   variation:= abs(getsol(x)-oldxval) ! Change in dx
  end-do
 end-procedure
```

With the initial guesses 0 for X and 1 for all B_t the model converges to an interest rate of 5.94413% ($x = 0.0594413$).

12.2 Goal Programming

Goal Programming is an extension of Linear Programming in which targets are specified for a set of constraints. In Goal Programming there are two basic models: the pre-emptive (lexicographic) model and the Archimedian model. In the pre-emptive model, goals are ordered according to priorities. The goals at a certain priority level are considered to be infinitely more important than the goals at the next level. With the Archimedian model weights or penalties for not achieving targets must be specified, and we attempt to minimize the sum of the weighted infeasibilities.

If constraints are used to construct the goals, then the goals are to minimize the violation of the constraints. The goals are met when the constraints are satisfied.

The example in this section demonstrates how Mosel can be used for implementing **pre-emptive Goal Programming with constraints**. We try to meet as many goals as possible, taking them in priority order.

12.2.1 Example problem

The objective is to solve a problem with two variables x and y, the constraint

$$100 \cdot x + 60 \cdot y \leq 600$$

and the three goal constraints

$$\begin{aligned} Goal_1 &: \quad 7 \cdot x + 3 \cdot y \geq 40 \\ Goal_2 &: \quad 10 \cdot x + 5 \cdot y = 60 \\ Goal_3 &: \quad 5 \cdot x + 4 \cdot y \geq 35 \end{aligned}$$

where the order given corresponds to their priorities.

12.2.2 Implementation

To increase readability, the implementation of the Mosel model is organized into three blocks: the problem is stated in the main part, procedure `preemptive` implements the solution strategy via preemptive Goal Programming, and procedure `print_sol` produces a nice solution printout.

```
model GoalCtr
 uses "mmxprs"

 forward procedure preemptive
 forward procedure print_sol(i:integer)

 declarations
  NGOALS=3                         ! Number of goals
  x,y: mpvar                       ! Decision variables
```

```
 dev: array(1..2*NGOALS) of mpvar  ! Deviation from goals
 MinDev: linctr                    ! Objective function
 Goal: array(1..NGOALS) of linctr  ! Goal constraints
end-declarations

100*x + 60*y <= 600               ! Define a constraint

! Define the goal constraints
Goal(1):=  7*x + 3*y >= 40
Goal(2):= 10*x + 5*y = 60
Goal(3):=  5*x + 4*y >= 35

preemptive                        ! Pre-emptive Goal Programming
```

At the end of the main part, we call procedure `preemptive` to solve this problem by pre-emptive Goal Programming. In this procedure, the goals are at first entirely removed from the problem ('hidden'). We then add them successively to the problem and re-solve it until the problem becomes infeasible, that is, the deviation variables forming the objective function are not all 0. Depending on the constraint type (obtained with `gettype`) of the goals, we need one (for inequalities) or two (for equalities) deviation variables.

Let us have a closer look at the first goal ($Goal_1$), a $\geq$ constraint: the left hand side (all terms with decision variables) must be at least 40 to satisfy the constraint. To ensure this, we add a dev_2. The goal constraint becomes $7 \cdot x + 3 \cdot y + dev_2 \geq 40$ and the objective function is to minimize dev_2. If this is feasible (0-valued objective), then we remove the deviation variable from the goal, thus turning it into a **hard constraint**. It is not required to remove it from the objective since minimization will always force this variable to take the value 0.

We then continue with $Goal_2$: since this is an equation, we need variables for positive and negative deviations, dev_3 and dev_4. We add $dev_4 - dev_3$ to the constraint, turning it into $10 \cdot x + 5 \cdot y + dev_4 - dev_3 = 60$ and the objective now is to minimize the absolute deviation $dev_4 + dev_3$. And so on.

```
procedure preemptive

! Remove (=hide) goal constraint from the problem
 forall(i in 1..NGOALS) sethidden(Goal(i), true)

 i:=0
 while (i<NGOALS) do
   i+=1
   sethidden(Goal(i), false)       ! Add (=unhide) the next goal

   case gettype(Goal(i)) of        ! Add deviation variable(s)
    CT_GEQ: do
             Deviation:= dev(2*i)
             MinDev += Deviation
            end-do
    CT_LEQ: do
             Deviation:= -dev(2*i-1)
             MinDev += dev(2*i-1)
            end-do
    CT_EQ : do
             Deviation:= dev(2*i) - dev(2*i-1)
             MinDev += dev(2*i) + dev(2*i-1)
            end-do
    else    writeln("Wrong constraint type")
            break
   end-case
   Goal(i)+= Deviation

   minimize(MinDev)                ! Solve the LP-problem
   writeln(" Solution(", i,"): x: ", getsol(x), ", y: ", getsol(y))
```

```
    if getobjval>0 then
     writeln("Cannot satisfy goal ",i)
     break
    end-if
    Goal(i)-= Deviation             ! Remove deviation variable(s) from goal
  end-do

  print_sol(i)                      ! Solution printout
 end-procedure
```

The procedure `sethidden(c:linctr, b:boolean)` has already been used in the previous chapter (Section 11.2) without giving any further explanation. With this procedure, constraints can be removed ('hidden') from the problem solved by the optimizer without deleting them in the problem definition. So effectively, the optimizer solves a **subproblem** of the problem originally stated in Mosel.

To complete the model, below is the procedure `print_sol` for printing the results.

```
 procedure print_sol(i:integer)
  declarations
   STypes={CT_GEQ, CT_LEQ, CT_EQ}
   ATypes: array(STypes) of string
  end-declarations

  ATypes:=[">=", "<=", "="]

  writeln(" Goal", strfmt("Target",11), strfmt("Value",7))
  forall(g in 1..i)
   writeln(strfmt(g,4), strfmt(ATypes(gettype(Goal(g))),4),
     strfmt(-getcoeff(Goal(g)),6),
     strfmt( getact(Goal(g))-getsol(dev(2*g))+getsol(dev(2*g-1)) ,8))

  forall(g in 1..NGOALS)
   if (getsol(dev(2*g))>0) then
    writeln(" Goal(",g,") deviation from target: -", getsol(dev(2*g)))
   elif (getsol(dev(2*g-1))>0) then
    writeln(" Goal(",g,") deviation from target: +", getsol(dev(2*g-1)))
   end-if
 end-procedure

end-model
```

When running the program, one finds that the first two goals can be satisfied, but not the third.

III. Working with the Mosel libraries

Whilst the two previous parts have shown how to work with the Mosel Language, this part introduces the programming language interface of Mosel in the form of the **Mosel C libraries**. The C interface is provided in the form of two libraries; it may especially be of interest to users who

- want to integrate models and/or solution algorithms written with Mosel into some larger system
- want to (re)use already existing parts of algorithms written in C
- want to interface Mosel with other software, for instance for graphically displaying results.

Other programming language interfaces available for Mosel are its **Java** and **Visual Basic** interfaces. They will be introduced with the help of small examples.

All these programming language interfaces only enable the user to access models, but not to modify them. The latter is only possible with the **Mosel Native Interface**. Even more importantly, the Native Interface makes it possible to add new constants, types, and subroutines to the Mosel Language. For more detail the reader is referred to the Native Interface user guide that is provided as a separate document. The Mosel Native Interface requires an additional licence.

Chapter 13

C interface

This chapter gives an introduction to the C interface of Mosel. It shows how to execute models from C and how to access modeling objects from C. It is not possible to make changes to Mosel modeling objects from C using this interface, but the data and parameters used by a model may be modified via files or run time parameters.

13.1 Basic tasks

To work with a Mosel model, in the C language or with the command line interpreter, it always needs to be compiled, then loaded into Mosel and executed. In this section we show how to perform these basic tasks in C.

13.1.1 Compiling a model in C

The following example program shows how Mosel is initialized in C, and how a model file (extension `.mos`) is compiled into a **bi**nary **m**odel (BIM) file (extension `.bim`). To use the Mosel Model Compiler Library, we need to include the header file `xprm_mc.h` at the start of the C program.

For the sake of readability, in this program, as for all others in this chapter, we only implement a rudimentary testing for errors.

```
#include <stdlib.h>
#include "xprm_mc.h"

int main()
{
 if(XPRMinit())                        /* Initialize Mosel */
  return 1;

 if(XPRMcompmod(NULL, "burglar2.mos", NULL, "Knapsack example"))
  return 2;                            /* Compile the model burglar2.mos,
                                          output the file burglar2.bim */
 return 0;
}
```

With version 1.4 of Mosel it becomes possible to redirect the BIM file that is generated by the compilation. Instead of writing it out to a physical file it may, for instance, be kept in memory or be written out in compressed format. The interested reader is refered to the whitepaper *Generalized file handling in Mosel*.

13.1.2 Executing a model in C

The example in this section shows how a Mosel binary model file (BIM) can be executed in C. The BIM file can of course be generated within the same program where it is executed, but here we leave out this step. A BIM file is an executable version of a model, but it does not include any data that is read in by the model from external files. It is portable, that is, it may be executed on a different type of architecture than the one it has been generated on. A BIM file produced by the Mosel compiler first needs to be loaded into Mosel (function `XPRMloadmod`) and can then be run by a call to function `XPRMrunmod`. To use these functions, we need to include the header file `xprm_rt.h` at the beginning of our program.

```
#include <stdio.h>
#include "xprm_rt.h"

int main()
{
 XPRMmodel mod;
 int result;

 if(XPRMinit())                       /* Initialize Mosel */
  return 1;

 if((mod=XPRMloadmod("burglar2.bim",NULL))==NULL)  /* Load a BIM file */
  return 2;

 if(XPRMrunmod(mod,&result,NULL))     /* Run the model */
  return 3;

 return 0;
}
```

The compile/load/run sequence may also be performed with a single function call to `XPRMexecmod` (in this case we need to include the header file `xprm_mc.h`):

```
#include <stdio.h>
#include "xprm_mc.h"

int main()
{
 int result;

 if(XPRMinit())                       /* Initialize Mosel */
  return 1;
                                      /* Execute = compile/load/run a model */
 if(XPRMexecmod(NULL,"burglar2.mos",NULL,&result,NULL))
  return 2;

 return 0;
}
```

13.2 Parameters

In Part I the concept of parameters in Mosel has been introduced: when a Mosel model is executed from the command line, it is possible to pass new values for its parameters into the model. The same is possible with a model run in C. If, for instance, we want to run model 'Prime' from Section 8.2 to obtain all prime numbers up to 500 (instead of the default value 100 set for the parameter `LIMIT` in the model), we may start a program with the following lines:

```
 XPRMmodel mod;
 int result;

 if(XPRMinit())                                /* Initialize Mosel */
  return 1;
```

```
 if((mod=XPRMloadmod("prime.bim",NULL))==NULL)  /* Load a BIM file */
  return 2;

 if(XPRMrunmod(mod,&result,"LIMIT=500"))        /* Run the model */
  return 3;
```

To use function XPRMexecmod instead of the compile/load/run sequence we have:

```
 int result;

 if(XPRMinit())                         /* Initialize Mosel */
  return 1;
                                        /* Execute with new parameter settings */
 if(XPRMexecmod(NULL,"prime.mos","LIMIT=500",&result,NULL))
  return 2;
```

13.3 Accessing modeling objects and solution values

Using the Mosel libraries, it is not only possible to compile and run models, but also to access information on the different modeling objects.

13.3.1 Accessing sets

A complete version of a program for running the model 'Prime' mentioned in the previous section may look as follows (we work with a model prime2 that corresponds to the one printed in Section 8.2 but with all output printing removed because we are doing this in C):

```
#include <stdio.h>
#include "xprm_mc.h"

int main()
{
 XPRMmodel mod;
 XPRMalltypes rvalue, setitem;
 XPRMset set;
 int result, type, i, size, first, last;

 if(XPRMinit())                                /* Initialize Mosel */
  return 1;

 if(XPRMexecmod(NULL,"prime2.mos","LIMIT=500",&result,&mod))
  return 2;                                    /* Execute the model */

 type=XPRMfindident(mod,"SPrime",&rvalue);    /* Get the object 'SPrime' */
 if((XPRM_TYP(type)!=XPRM_TYP_INT)||          /* Check the type: */
    (XPRM_STR(type)!=XPRM_STR_SET))           /* it must be a set of integers */
  return 3;
 set = rvalue.set;

 size = XPRMgetsetsize(set);                   /* Get the size of the set */
 if(size>0)
 {
  first = XPRMgetfirstsetndx(set);             /* Get number of the first index */
  last = XPRMgetlastsetndx(set);               /* Get number of the last index */
  printf("Prime numbers from 2 to %d:\n", LIM);
  for(i=first;i<=last;i++)                     /* Print all set elements */
   printf(" %d,",XPRMgetelsetval(set,i,&setitem)->integer);
  printf("\n");
 }
```

```
 return 0;
}
```

To print the contents of set `SPrime` that contains the desired result (prime numbers between 2 and 500), we first retrieve the Mosel reference to this object using function `XPRMfindident`. We are then able to enumerate the elements of the set (using functions `XPRMgetfirstsetndx` and `XPRMgetlastsetndx`) and obtain their respective values with `XPRMgetelsetval`.

13.3.2 Retrieving solution values

The following program executes the model 'Burglar3' (the same as model 'Burglar2' from Chapter 2 but with all output printing removed) and prints out its solution.

```
#include <stdio.h>
#include "xprm_rt.h"

int main()
{
 XPRMmodel mod;
 XPRMalltypes rvalue, itemname;
 XPRMarray varr, darr;
 XPRMmpvar x;
 XPRMset set;
 int indices[1], result, type;
 double val;

 if(XPRMinit())                          /* Initialize Mosel */
  return 1;

 if((mod=XPRMloadmod("burglar3.bim",NULL))==NULL)  /* Load a BIM file */
  return 2;

 if(XPRMrunmod(mod,&result,NULL))        /* Run the model (includes
                                            optimization) */
  return 3;

 if((XPRMgetprobstat(mod)&XPRM_PBRES)!=XPRM_PBOPT)
  return 4;                              /* Test whether a solution is found */

 printf("Objective value: %g\n", XPRMgetobjval(mod));
                                         /* Print the obj. function value */

 type=XPRMfindident(mod,"take",&rvalue);   /* Get the model object 'take' */
 if((XPRM_TYP(type)!=XPRM_TYP_MPVAR)||     /* Check the type: */
    (XPRM_STR(type)!=XPRM_STR_ARR))        /* it must be an 'mpvar' array */
  return 5;
 varr = rvalue.array;

 type=XPRMfindident(mod,"VALUE",&rvalue);  /* Get the model object 'VALUE' */
 if((XPRM_TYP(type)!=XPRM_TYP_REAL)||      /* Check the type: */
    (XPRM_STR(type)!=XPRM_STR_ARR))        /* it must be an array of reals */
  return 6;
 darr = rvalue.array;

 type=XPRMfindident(mod,"ITEMS",&rvalue);  /* Get the model object 'ITEMS' */
 if((XPRM_TYP(type)!=XPRM_TYP_STRING)||    /* Check the type: */
    (XPRM_STR(type)!=XPRM_STR_SET))        /* it must be a set of strings */
  return 7;
 set = rvalue.set;

 XPRMgetfirstarrentry(varr, indices);      /* Get the first entry of array varr
                                              (we know that the array is dense
```

```
                                            and has a single dimension) */
 do
 {
  XPRMgetarrval(varr,indices,&x);          /* Get a variable from varr */
  XPRMgetarrval(darr,indices,&val);        /* Get the corresponding value */
  printf("take(%s): %g\t (item value: %g)\n", XPRMgetelsetval(set, indices[0],
         &itemname)->string, XPRMgetvsol(mod,x), val);
                                           /* Print the solution value */
 } while(!XPRMgetnextarrentry(varr, indices));  /* Get the next index tuple */

 return 0;
}
```

The array of variables varr is enumerated using the array functions XPRMgetfirstarrentry and XPRMgetnextarrentry. These functions may be applied to arrays of any type and dimension (for higher numbers of dimensions, merely the size of the array indices that is used to store the index-tuples has to be adapted). With these functions we run systematically through all possible combinations of index tuples, hence the hint at **dense** arrays in the example. In the case of sparse arrays it is preferrable to use different enumeration functions that only enumerates those entries that are defined (see next section).

13.3.3 Sparse arrays

In Chapter 3 the problem 'Transport' has been introduced. The objective of this problem is to calculate the flows $flow_{pr}$ from a set of plants to a set of sales regions that satisfy all demand and supply constraints and minimize the total cost. Not all plants may deliver goods to all regions. The flow variables $flow_{pr}$ are therefore defined as a **sparse** array. The following example prints out all existing entries of the array of variables.

```
#include <stdio.h>
#include "xprm_rt.h"

int main()
{
 XPRMmodel mod;
 XPRMalltypes rvalue;
 XPRMarray varr;
 XPRMset *sets;
 int *indices, dim, result, type, i;

 if(XPRMinit())                         /* Initialize Mosel */
  return 1;

 if((mod=XPRMloadmod("transport.bim",NULL))==NULL)  /* Load a BIM file */
  return 2;

 if(XPRMrunmod(mod,&result,NULL))       /* Run the model */
  return 3;

 type=XPRMfindident(mod,"flow",&rvalue); /* Get the model object named 'flow' */
 if((XPRM_TYP(type)!=XPRM_TYP_MPVAR)||   /* Check the type: */
    (XPRM_STR(type)!=XPRM_STR_ARR))      /* it must be an array of unknowns */
  return 4;
 varr=rvalue.array;

 dim = XPRMgetarrdim(varr);             /* Get the number of dimensions of
                                           the array */
 indices = (int *)malloc(dim*sizeof(int));
 sets = (XPRMset *)malloc(dim*sizeof(XPRMset));

 XPRMgetarrsets(varr,sets);             /* Get the indexing sets  */
 XPRMgetfirstarrtruentry(varr,indices);  /* Get the first true index tuple */
```

```
 do
 {
  printf("flow(");
  for(i=0;i<dim-1;i++)
   printf("%s,",XPRMgetelsetval(sets[i],indices[i],&rvalue)->string);
  printf("%s), ",XPRMgetelsetval(sets[dim-1],indices[dim-1],&rvalue)->string);
 } while(!XPRMgetnextarrtruentry(varr,indices));  /* Get next true index tuple*/
 printf("\n");

 free(sets);
 free(indices);

 return 0;
}
```

In this example, we first get the number of indices (dimensions) of the array of variables varr (using function XPRMgetarrdim). We use this information to allocate space for the arrays sets and indices that will be used to store the indexing sets and single index tuples for this array respectively. We then read the indexing sets of the array (function XPRMgetarrsets) to be able to produce a nice printout.

The enumeration starts with the first defined index tuple, obtained with function XPRMgetfirstarrtruentry, and continues with a series of calls to XPRMgetnextarrtruentry until all defined entries have been enumerated.

13.3.4 Problem solving in C with Xpress-Optimizer

In certain cases, for instance if the user wants to re-use parts of algorithms that he has written in C for the Xpress-Optimizer, it may be necessary to pass from a problem formulation with Mosel to solving the problem in C by direct calls to the Xpress-Optimizer. The following example shows how this may be done for the Burglar problem. We use a slightly modified version of the original Mosel model:

```
model Burglar4
 uses "mmxprs"

 declarations
  WTMAX=102                          ! Maximum weight allowed
  ITEMS={"camera", "necklace", "vase", "picture", "tv", "video",
         "chest", "brick"}           ! Index set for items

  VALUE: array(ITEMS) of real        ! Value of items
  WEIGHT: array(ITEMS) of real       ! Weight of items

  take: array(ITEMS) of mpvar        ! 1 if we take item i; 0 otherwise
 end-declarations

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

! Objective: maximize total value
 MaxVal:= sum(i in ITEMS) VALUE(i)*take(i)

! Weight restriction
 sum(i in ITEMS) WEIGHT(i)*take(i) <= WTMAX

! All variables are 0/1
 forall(i in ITEMS) take(i) is_binary

 setparam("XPRS_LOADNAMES", true)    ! Enable loading of object names
 loadprob(MaxVal)                    ! Load problem into the optimizer

end-model
```

The procedure `maximize` to solve the problem has been replaced by `loadprob`. This procedure loads the problem into the optimizer without solving it. We also enable the loading of names from Mosel into the optimizer so that we may obtain an easily readable output.

The following C program reads in the Mosel model and solves the problem by direct calls to Xpress-Optimizer. To be able to address the problem loaded into the optimizer, we need to retrieve the optimizer problem pointer from Mosel. Since this information is a parameter (`XPRS_PROBLEM`) that is provided by module `mmxprs`, we first need to obtain the reference of this library (by using function `XPRMfinddso`).

```
#include <stdio.h>
#include "xprm_rt.h"
#include "xprs.h"

int main()
{
 XPRMmodel mod;
 XPRMdsolib dso;
 XPRSprob prob;
 int result, ncol, len, i;
 double *sol, val;
 char *names;

  if(XPRMinit())                         /* Initialize Mosel */
  return 1;

 if((mod=XPRMloadmod("burglar4.bim",NULL))==NULL)  /* Load a BIM file */
  return 2;

 if(XPRMrunmod(mod,&result,NULL))       /* Run the model (no optimization) */
  return 3;

  /* Retrieve the pointer to the problem loaded in the Xpress-Optimizer */
 if((dso=XPRMfinddso("mmxprs"))==NULL)
  return 4;
 if(XPRMgetdsoparam(mod, dso, "XPRS_PROBLEM", &result, (XPRMalltypes *)&prob))
  return 5;

 if(XPRSmaxim(prob, "g"))               /* Solve the problem */
  return 6;

 if(XPRSgetintattrib(prob, XPRS_MIPSTATUS, &result))
  return 7;
                                       /* Test whether a solution is found */
 if((result==4) || (result==6))
 {
  if(XPRSgetdblattrib(prob, XPRS_MIPOBJVAL, &val))
   return 8;
  printf("Objective value: %g\n", val); /* Print the objective function value */

  if(XPRSgetintattrib(prob, XPRS_COLS, &ncol))
   return 9;
  if((sol = (double *)malloc(ncol * sizeof(double)))==NULL)
   return 10;
  if(XPRSgetsol(prob, sol, NULL, NULL, NULL))
   return 11;                          /* Get the primal solution values */
  if(XPRSgetintattrib(prob, XPRS_NAMELENGTH, &len))
   return 11;                          /* Get the maximum name length */
  if((names = (char *)malloc((len*8+1)*ncol*sizeof(char)))==NULL)
   return 12;
  if(XPRSgetnames(prob, 2, names, 0, ncol-1))
   return 13;                          /* Get the variable names */

  for(i=0; i<ncol; i++)                /* Print out the solution */
    printf("%s: %g\n", names+((len*8+1)*i), sol[i]);
```

```
  free(names);
  free(sol);
 }
 return 0;
}
```

Since the Mosel language provides ample programming facilities, in most applications there will be no need to switch from the Mosel language to problem solving in C. If nevertheless this type of implementation is chosen, it should be noted that it is not possible to get back to Mosel, once the Xpress-Optimizer has been called directly from C: the solution information and any possible changes made to the problem directly in the optimizer are not communicated to Mosel.

Chapter 14

Other programming language interfaces

In this chapter we show how the examples from Sections 13.1 and 13.2 may be written with other programming languages, namely Java and Visual Basic.

14.1 Java

To use the Mosel Java classes the line `import com.dashoptimization.*;` must be added at the beginning of the program.

14.1.1 Compiling and executing a model in Java

With Java it is not required to initialize Mosel explicitly. To execute a Mosel model in Java we merely need to call the three Mosel functions performing the standard compile/load/run sequence as shown in the following example.

```
import java.io.*;
import com.dashoptimization.*;

public class ugcomp
{
 public static void main(String[] args) throws java.io.IOException
 {
  XPRMmodel mod;

  System.out.println("Compiling ‘burglar2’");
  XPRM.compile("burglar2.mos");

  System.out.println("Loading ‘burglar2’");
  mod = XPRM.loadModel("burglar2.bim");

  System.out.println("Executing ‘burglar2’");
  mod.run();

  System.out.println("‘burglar2’ returned: " + mod.getResult());
 }
}
```

14.1.2 Parameters

When executing a Mosel model in Java, it is possible to pass new values for its parameters into the model. If, for instance, we want to run model ‘Prime’ from Section 8.2 to obtain all prime numbers up to 500 (instead of the default value 100 set for the parameter `LIMIT` in the model), we may write the following program:

```
import java.io.*;
import com.dashoptimization.*;

public class ugparam
{
 public static void main(String[] args) throws java.io.IOException
 {
  XPRMmodel mod;

  System.out.println("Compiling ‘prime’");
  XPRM.compile("prime.mos");

  System.out.println("Loading ‘prime’");
  mod = XPRM.loadModel("prime.bim");

  System.out.println("Executing ‘prime’");
  mod.run("LIMIT=500");

  System.out.println("‘prime’ returned: " + mod.getResult());
 }
}
```

14.2 Visual Basic

In Visual Basic, a Mosel model needs to be embedded into a project. In this section we shall only show the parts relevant to the Mosel functions, assuming that the execution of a model is trigged by the action of clicking on some object.

14.2.1 Compiling and executing a model in Visual Basic

As with the other programming languages, to execute a Mosel model in Visual Basic we need to perform the standard compile/load/run sequence as shown in the following example. We use a slightly modified version `burglar5.mos` of the burglar problem where we have redirected the output printing to the file `burglar_out.txt`.

```
Private Sub burglar_Click()
  Dim model As Long
  Dim ret As Long
  Dim result As Long

'Initialize Mosel
  ret = XPRMinit
  If ret <> 0 Then
    MsgBox "Initialization error (" & ret & ")"
    Exit Sub
  End If

'Compile burglar5.mos
  ret = XPRMcompmod(vbNullString, "burglar5.mos", vbNullString, "Knapsack")
  If ret <> 0 Then
    MsgBox "Compile error (" & ret & ")"
    Exit Sub
  End If

'Load burglar5.bim
  model = XPRMloadmod("burglar5.bim", vbNullString)
  If model = 0 Then
    MsgBox "Error loading model"
    Exit Sub
  End If
```

```
'Run the model
  ret = XPRMrunmod(model, result, vbNullString)
  If ret <> 0 Then
    MsgBox "Execution error (" & ret & ")"
    Exit Sub
  End If
End Sub
```

14.2.2 Parameters

When executing a Mosel model in Visual Basic, it is possible to pass new values for its parameters into the model. The following program extract shows how we may run model 'Prime' from Section 8.2 to obtain all prime numbers up to 500 (instead of the default value 100 set for the parameter `LIMIT` in the model). We use a slightly modified version `prime3.mos` of the model where we have redirected the output printing to the file `prime_out.txt`.

```
Private Sub prime_Click()
  Dim model As Long
  Dim ret As Long
  Dim result As Long

'Initialize Mosel
  ret = XPRMinit
  If ret <> 0 Then
    MsgBox "Initialization error (" & ret & ")"
    Exit Sub
  End If

'Compile prime3.mos
  ret = XPRMcompmod(vbNullString, "prime3.mos", vbNullString, "Prime numbers")
  If ret <> 0 Then
    MsgBox "Compile error (" & ret & ")"
    Exit Sub
  End If

'Load prime3.bim
  model = XPRMloadmod("prime3.bim", vbNullString)
  If model = 0 Then
    MsgBox "Error loading model"
    Exit Sub
  End If

'Run model with new parameter settings
  ret = XPRMrunmod(model, result, "LIMIT=500")
  If ret <> 0 Then
    MsgBox "Execution error (" & ret & ")"
    Exit Sub
  End If
End Sub
```

Index