Using ODBC and other database interfaces with Mosel

Data exchange with spreadsheets and databases

6.10

WHITEPAPER

FICO® Xpress Optimization

FICO

FICO® Xpress Mosel 6.10 (FICO® Xpress 9.7)
Last Revised: 29 July, 2025

How to Contact the Xpress Team

Sales and Maintenance

If you need information on other Xpress Optimization products, or you need to discuss maintenance contracts or other sales-related items, contact FICO by:

- Phone: +1 (408) 535-1500 or +44 207 940 8718
- Web: www.fico.com/optimization and use the available contact forms

Product Support

Customer Self Service Portal (online support): www.fico.com/en/product-support

Email: Support@fico.com (Please include *'Xpress'* in the subject line)

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

Using ODBC and other database interfaces with Mosel

Data exchange with spreadsheets and databases

S. Heipcke

Xpress Optimization, FICO, 6280 Bishops Court, Birmingham Business Park, Birmingham B37 7YB, UK
http://www.fico.com/xpress

Release 6.10

29 July, 2025

Abstract

This document gives an introduction to the available Mosel interfaces for accessing databases and spreadsheets from within a model. It provides a large number of examples illustrating topics such as sparse and dense data formats, tables holding entries of several data arrays, data arrays read in from several tables, use of the data structures 'list' and 'record', and the handling of dates and time data.

The ODBC protocol is supported by many database software products. Data may be transfered between a model and an external (ODBC) data source through `initializations` blocks, using the *odbc* I/O driver, or with the help of SQL commands. The latter option provides greater flexibility but requires some knowledge of the SQL language.

The software-specific interface for working with Oracle databases defined by the module *mmoci* is very similar to the ODBC interface (I/O driver and SQL)—the differences are explained by indicating the necessary modifications to the examples.

This paper also discusses the spreadsheet interfaces defined by the module *mmsheet*, namely the direct, software-specific access to Excel spreadsheets via the I/O driver *excel*, the more generic *xls* and *xlsx* drivers, and the interface to CSV format data.

Contents

1 Introduction

The Mosel distribution contains several different interfaces for exchanging data between a model and a database or spreadsheet. As well as an ODBC interface there are software-specific implementations for Oracle and MS Excel, all of which are presented in this document.

ODBC is a protocol for working with databases as external data sources. It can also be used to access data in spreadsheets such as certain versions of MS Excel. However, with spreadsheets some restrictions apply since spreadsheets do not implement the full functionality of databases, especially for writing data into them. As an alternative to ODBC therefore different spreadsheet interfaces (supporting MS Excel formats) are available that remedy some of these drawbacks, they should be used for accessing data in spreadsheets in place of the ODBC interface.

The Mosel module *mmodbc* provides access to ODBC functionality from within Mosel models. As always with Mosel modules, when we wish to use this module in a model, we need to indicate its name in a `uses` statement at the beginning of our model, immediately after the model name:

```
uses "mmodbc"
```

The reader will find the complete documentation of the Mosel module *mmodbc* in the 'Mosel Language Reference Manual', Chapter 'mmodbc'.

A separate module, *mmoci*, defines a software-specific interface to Oracle databases (OCI). The functionality and manner of use of this module closely resembles that of the ODBC module. The `uses` statement for working with OCI is:

```
uses "mmoci"
```

The OCI module is documented in the 'Mosel Language Reference Manual', Chapter 'mmoci'.

The spreadsheet and CSV interfaces are provided by the Mosel module *mmsheet* that is equally documented in the 'Mosel Language Reference Manual', Chapter 'mmsheet'. The corresponding `uses` statement for this module is:

```
uses "mmsheet"
```

The aim of the present document is to explain the different features of the ODBC module (and by analogy, the OCI module) by means of a collection of examples. In the beginning we show how to

- set up ODBC,
- work with the *odbc* I/O driver in `initializations` blocks,
- use the full functionality of the module *mmodbc* in the formulation of SQL queries,
- access MS Excel spreadsheets via the dedicated spreadsheet I/O drivers, and
- set module parameters that may be helpful for debugging.

The main part of this document describes a number of examples that illustrate topics such as

- sparse *vs.* dense data format,
- reading from / writing to tables holding entries of several data arrays,
- data arrays read from several tables,
- defining SQL queries,
- using the data structures 'list' and 'record',
- working with date and time data types.

All examples described in this document are provided as part of the Xpress distribution (subdirectory `examples/mosel/Whitepapers/MoselData`). Most examples have six versions, namely (1) using an ODBC connection with standard Mosel data initializations to access databases, (2) using SQL statements with the same data sources, (3) using the dedicated Excel interface (driver *excel*), (4) using the Oracle interface, (5) using the generic Excel interface (drivers *xls/xlsx*), and (6) using the CSV interface.

The last section contains some hints on how to detect and solve typical problems that may occur when working with ODBC.

2 Software setup

2.1 Setting up ODBC

The ODBC technology is available for many database/platform combinations. It relies on a *driver manager* that is used as an interface between applications (like *mmodbc*) and a *data source* that is accessed through a dedicated *driver*. The Mosel module *mmodbc* provides an interface to ODBC, it does *not* contain any drivers or driver managers. These must therefore be installed and set up on the operating system before this module can be used.

Under Windows, usually the driver manager is part of the system and most data sources (*e.g.*, MS Access, MS Excel, SQLserver) are provided with their ODBC driver. The ODBC drivers are set up as *User Data Source* in the *ODBC Data Source Administrator*. To check which drivers are set up under Windows (2000 or more recent) select *Start* ≫ *Settings* ≫ *Control Panel* ≫ *Administrative Tools* ≫ *Data Sources (ODBC)*. For example, for MS Access there should be a data source named `MS Access Database` (this name is refered to in ODBC as *DSN*, the data source name). If the drivers are not set up, you may add the DSN for Access by clicking *Add* and selecting *Microsoft Access Driver (*.mdb, *.accdb)*.

On the other supported operating systems it may be necessary to install a driver manager as well as the required driver(s). The module *mmodbc* supports two driver managers: *iODBC* (http://www.iodbc.org) and *unixODBC* (http://www.unixodbc.org). The module initialization succeeds only if one of these two driver managers is installed and can be accessed (in general this requires updating some environment variables). In addition, you must make sure that the ODBC driver for the data source you wish to use is installed. For the database MySQL, for instance, you can download the required ODBC driver, MyODBC, from http://dev.mysql.com/downloads/connector/odbc.

2.1.1 ODBC connection strings in Mosel

When connecting to a database from a Mosel model the filename is given in the form of a *connection string* that consists of the DSN and the name of the external data source (abbreviated as *DBQ* (Windows) or *DB*), such as `'DSN=mysql;DB=data'` or `'DSN=MS Access Database;DBQ=C:/xpress/examples/data.mdb'`. Please note that some databases do not accept blank spaces in the connection string.

Under Windows it is not necessary to state explicitely the DSN in the connection string since Mosel will automatically locate the appropriate driver (if it is installed). We may therefore work with connection strings shortened to the name of the external data source, such as `'data.mdb'`.

For a more general introduction to the concept of database connection strings the reader is refered to http://www.connectionstrings.com.

2.2 The Excel interface

The spreadsheet interfaces defined by the module *mmsheet* do not require any extra setup or additional software as this is the case with ODBC. The dedicated Excel driver *excel* can only be used if MS Excel is installed and licensed. The drivers *xls*, *xlsx*, and *csv* are independent of Excel and can be used, including, on non-Windows platforms.

Excel spreadsheets are accessed by simply stating their file name, preceded by the driver prefix as shown below (see Section 3.2).

Although certain spreadsheets support some ODBC functionality, we recommend to use one of the dedicated spreadsheet interfaces in the place of an ODBC connection. The access to the spreadsheet is more direct and hence more efficient and these interfaces remove some restrictions and possible sources of problems specific to the use of ODBC technology with Excel spreadsheets.

2.3 The Oracle interface

The Oracle interface defined by the module *mmoci* accesses Oracle databases via the Oracle Call Interface (OCI). Oracle's Instant Client package must be installed on the machine that runs the Mosel model. The Oracle Instant Client package is available for download from https://www.oracle.com/database/technologies/instant-client/downloads.html

The *logon information* for an Oracle database comprises the user name and password along with the database name, formated as a string such as `'myusername/mypassword@dbname'`.

It is possible to access Oracle databases via an ODBC connection (that is, using module *mmodbc* instead of *mmoci*). In this case, an appropriate ODBC driver must be installed.

2.4 The SQLite interface

SQLite databases can be accessed via the ODBC interface. SQLite is included in the *mmodbc* module on all platforms that are supported by Xpress. The ODBC driver *mmsqlite* to access SQLite equally forms part of the distribution, **no ODBC setup** and no additional installations are required when using this driver. The full connection string to employ in this case has the form `'DRIVER=mmsqlite;READONLY=false;DB=mydatabase.sqlite'` (notice that instead of referring to a DSN definition we directly use the built-in ODBC driver). However, it is sufficient to simply state the database filename (and path) if it has one of the extensions `sqlite`, `sqlite3`, `db`, or `db3` — Mosel will automatically generate the complete connection string.

Alternatively, if you wish to use a different version of SQLite than the one provided with Mosel, you need to follow the ODBC installation procedure: after downloading an external ODBC driver for SQLite, a DSN must be setup as explained above in Section 2.1. Assuming that we have called the DSN 'sqlite', this results in a connection string of the form `'DSN=sqlite;DATABASE=mydatabase.sqlite'`.

For visualizing and editing SQLite databases, you may choose to install additional software, such as the SQLite Manager plugin for Firefox browsers that can be downloaded from https://addons.mozilla.org or the DB Browser for SQLite at http://sqlitebrowser.org

3 Introductory example

The standard Mosel syntax for reading and writing data uses `initializations` blocks to access external files, such as

```
declarations
 A: array(set of integer) of real   ! Array of unknown size (=dynamic)
 B: array(1..7) of string           ! Array of known size (=static)
end-declarations

initializations from "mydata.dat"
 A
 B as "MyB"
end-initializations
```

where the datafile `mydata.dat` may have the following contents:

```
A: [(3) 2  (1) 4.2  (6) 9  (10) 7.5  (-1) 3]
MyB: ["Mon" "Tue" "Wed" "Thu" "Fri" "Sat" "Sun"]
```

To obtain access to file types other than text files in Mosel format we merely need to modify the filename string, prefixing the name of the data source by the I/O driver we want to use followed by a colon. For instance, to read a text file in comma separated format we may use the driver prefix

"mmetc.diskdata:". I/O drivers may be seen as filters that decode data from some other format, transforming it into the format used by Mosel—or the other way round. For further detail on the concept of I/O drivers, the reader is referred to the Xpress Whitepaper *Generalized file handling in Mosel*.

3.1 ODBC

To access spreadsheets or databases through `initializations` blocks using an ODBC connection we need to prefix the name of the data source (ODBC connection string as described in Section 2.1.1) by the name of the ODBC I/O driver followed by a colon, that is, "mmodbc.odbc:".

3.1.1 Data input using *odbc*

As a first example we shall now see how to read data from an MS Access database into a Mosel array. Let us suppose that in a database called `data.mdb` you have created the following table `MyDataTable` with 3 fields holding the following data values:

Index_i	Index_j	Value
1	1	12.5
2	3	5.6
10	9	-7.1
3	2	1

We may then use the following model `duo.mos` to read in the array `A4` from the database and print it out.

```
model "Duo input (1)"
 uses "mmodbc"

 declarations
  A4: dynamic array(range,range) of real
 end-declarations

! Use an initializations block with the odbc driver to read data
 initializations from "mmodbc.odbc:data.mdb"
  A4 as 'MyDataTable'
 end-initializations

! Print out the data we have read
 writeln('A4 is: ', A4)

end-model
```

If we want to read the data from another database, say the SQLite database `data.sqlite`, the only change we need to make is to change the filename string to "mmodbc.odbc:data.sqlite". For a MySQL database `data` we would have to use the long form of the connection string: "mmodbc.odbc:DSN=mysql;DB=data". Any database we use with the model printed above needs to contain a table called 'MyDataTable' with three fields, the first two (for the indices) of type `integer`, and the third of type `double`.

3.1.2 Data output using *odbc*

Outputting data from a Mosel model through the *odbc* I/O driver again only requires few changes to the model. Consider the following example (file `duo_out.mos`)—notice that the index ranges are -1, 0, 1 and 5, 6, 7 and not the standard 1, 2, 3:

```
model "Duo output (1)"
 uses "mmodbc"

 declarations
  A: array(-1..1,5..7) of real
 end-declarations

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

! Use an initializations block with the odbc driver for writing data
 initializations to "mmodbc.odbc:data.mdb"
  A as "MyOutTable1"
 end-initializations

end-model
```

When we wish to write data to a database we need to prepare a suitable data table to receive the data: our table needs to be named 'MyOutTable1' with fields that correspond to the data array we want to write. In our case, the first two (index) fields must be of type `integer` and the third field of type `double`.

In terms of database functionality, when writing out data with `initializations to`, Mosel performs an "insert", no replacement or update. If the data table contains already some data, for instance from previous model runs, the new output will be appended to the existing data.

In the case of a database table, the insertion will fail if a key field has been defined and you are trying to write the same data entries a second time. The deletion of any existing data in the table(s) used for output must be done manually directly in the database or spreadsheet, or with the corresponding SQL commands in the Mosel model (the latter option only applies to databases). With this objective, the *odbc* I/O driver may be used in combination with other *mmodbc* functionality, for instance to execute specific SQL queries (see Section 4.1).

3.2 Spreadsheets: Excel and CSV

MS Excel spreadsheets can be accessed directly from a Mosel model with the help of the *excel* or *xls/xlsx* I/O drivers. The use of these drivers is similar to what we have seen above for the *odbc* driver.

Yet another method for accessing spreadsheet data consists in using the CSV driver *csv* with spreadsheets that have been saved in CSV format. All the dedicated spreadsheet drivers are defined by the module *msheet*. The main differences between the functionality and usage of the various spreadsheet drivers are summarized in the following table.

The *diskdata* driver defined in the module *mmetc* also provides functionality for reading and writing CSV format text files. For the items listed in Table 1 it has the same charateristics as the *csv* driver. The requirements on the layout of the data file by *diskdata* are somewhat more restrictive than those made by the *csv* I/O driver—most importantly, this driver expects a single data table per file. The *diskdata* driver works in a linewise fashion (always reading all data rows of a file), as opposed to the I/O drivers of the *mmsheet* module that load the entire data file into memory when first accessing it and then allow the user to select specific ranges. Its use may be preferrable to *csv* with respect to memory use and performance when reading large data files, particularly on platforms where loading large files is time-consuming.

3.2.1 Data input using the *excel* or *xls* I/O drivers

We shall work with a spreadsheet that has the following contents:

Table 1: Comparison of spreadsheet I/O drivers

	excel	*xls/xlsx*	*csv*
File type	physical file	physical file	extended file
Supported platforms	Windows	Windows, Linux, macOS	all Xpress platforms
Requirements	Excel + open interactive session	none, can be used remotely	none, can be used remotely
File creation for output	no	yes	yes
Output writing mechanism	on-screen display without saving if application running, otherwise data saved into file	data saved into file	data saved into file
Named ranges	yes	yes	no
Multiple worksheets	yes	yes	no
VBA macros	yes	no	no

	A	B	C	D	E
1					
2		**Index_i**	**Index_j**	**Value**	
3		1	1	12.5	
4		2	3	5.6	
5		10	9	-7.1	
6		3	2	1	
7					

Assuming that we have named 'MyDataTable2' the cell range B3:D6 (that is, selecting just the data, excluding the header row) we can then read in these data with the following model `duoexc.mos`.

```
model "Duo input (Excel)"
 uses "mmsheet"

 declarations
  A4: dynamic array(range,range) of real
 end-declarations

! Use the excel driver for reading the data
 initializations from "mmsheet.excel:data.xls"
  A4 as 'MyDataTable2'
 end-initializations

! Print out the data we have read
 writeln('A4 is: ', A4)

end-model
```

It is also possible to read the data from a range 'MyDataTable' that inludes the header row (that is, selecting the area B2:D6). In this case we need to specify the driver option `skiph` to skip the header line of the selected data range:

```
 initializations from "mmsheet.excel:data.xls"
  A4 as 'skiph;MyDataTable'
 end-initializations
```

mmsheet

Yet another possibility is to use directly the worksheet and cell references instead of defining a named range (NB: the first sheet of the workbook is selected if no worksheet name is specified and the used cells of the selected sheet are assumed if no cell range selection is provided):

```
 initializations from "mmsheet.excel:data.xls"
  A4 as '[Sheet1$B3:D6]'
 end-initializations
```

Working with named ranges has the advantage over this explicit form that modifications to the spreadsheet layout repositioning the data range will not make it necessary to modify the model.

Instead of the *excel* driver that can only be used with an existing MS Excel installation, we can switch to the generic *xls* driver by modifying the file name to

```
 "mmsheet.xls:data.xls"
```

All else, including the driver options and the usage of named ranges, remains unchanged.
Note: with a spreadsheet saved in XLSX format (files with the extension `.xlsx`) we would have to use the driver *xlsx*.

3.2.2 Data output using *excel* or *xls*

The following model `duoexc_out.mos` writes out the array `A` into the spreadsheet range F3:H3 that we have called 'MyOutTable3'. These cells denote the first row of the rectangular area into which we wish to write.

```
model "Duo output (Excel)"
 uses "mmsheet"

 declarations
  A: array(-1..1,5..7) of real
 end-declarations

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

! Use an initializations block with the excel driver for writing data
 initializations to "mmsheet.excel:data.xls"
  A as "grow;MyOutTable3"
 end-initializations

end-model
```

In this model we have used the option `grow` of the *excel* driver to indicate that the actual output area may grow (= add more rows, the number of selected columns must be sufficient) beyond the specified output range as required by the data. Alternatively, we may also specify the complete output range such as in

```
 initializations to "mmsheet.excel:data.xls"
  A as '[Sheet1$F3:H11]'
 end-initializations
```

or more dynamically:

```
 initializations to "mmsheet.excel:data.xls"
  A as '[Sheet1$F3:H' + (3+A.size-1) + ']'
 end-initializations
```

If the output range has been defined to include the header row (ODBC-compatible format) we need to use again the option `skiph`.

```
initializations to "mmsheet.excel:data.xls"
 A as 'skiph;grow;MyOutTable1'
end-initializations
```

When using the *excel* or *xls/xlsx* drivers the definition of the output range in the spreadsheet remains unchanged even if the actual output area exceeds its length. As a consequence, the output from a second model run will start at exactly the same place as the first, overwriting any previous results in the same location (but not deleting any lines if the output from the second run uses fewer rows than the first).

A specific feature of the *excel* driver is that the Excel spreadsheet file may remain open while writing to it from Mosel. In this case the data written to the spreadsheet does not get saved, enabling the user thus to experiment with model runs without causing any unwanted lasting effects to the output file. However, when using the *xls/xlsx* drivers, output data is saved directly into the spreadsheet file—the output file needs to be closed if the application used for displaying it locks write access to the file by other programs.

With the *excel* driver, the output file must exist prior to writing to it from Mosel. The *xls/xlsx* drivers will create a new spreadsheet file of the corresponding format if the specified file is not found, in which case it is obviously not possible to work with predefined ranges, but the option `skiph+` can be used to output a header line, for example as in the following version that writes output into the columns F to H of the first sheet of a new spreadsheet file 'anewfile.xls':

```
initializations to "mmsheet.xls:anewfile.xls"
 A as 'skiph+;[1$F:H](Index1,Index2,Value_of_A)'
end-initializations
```

3.2.3 Data input using the *csv* I/O driver

Now assume that the spreadsheet from Section 3.1.1 has been saved in CSV format into the file `data.csv`. We can then read in these data with the following model `duosheet.mos`.

```
model "Duo input (CSV)"
 uses "mmsheet"

 declarations
  A6: dynamic array(range,range) of real
 end-declarations

! Use the csv driver for reading the data
 initializations from "mmsheet.csv:data.csv"
  A6 as '[B3:D6]'
 end-initializations

! Print out the data we have read
 writeln('A6 is: ', A6)

end-model
```

The CSV format does not support the definition of named ranges and there is no notion of 'worksheet' (when saving an Excel spreadsheet in CSV format the user selects the sheet to be saved). We therefore need to address cell ranges explicitly by indicating their position using the (letter,number) notation as shown above or alternatively, with RC (row-column) notation:

```
initializations from "mmsheet.csv:data.csv"
 A6 as '[R3C2:R6C4]'
```

```
 end-initializations
```

Driver options such as `skiph` to skip the range header line apply as before

```
 initializations from "mmsheet.csv:data.xls"
  A4 as 'skiph;[B2:D6]'
 end-initializations
```

As for the spreadsheet drivers, it is also possible to use an empty range definition, in which case all contents of the CSV file is considered as the selected area.

```
 initializations from "mmsheet.csv:data.csv"
   A6 as '[]'
 end-initializations
```

If we want to read all the content of columns B to D, skipping the first line, we can use the following shorthand notation (NB: this notation without row numbers, just like the row-column notation works with all *mmsheet* drivers):

```
 initializations from "mmsheet.csv:data.xls"
  A4 as 'skiph;[B:D]'
 end-initializations
```

The main advantage of CSV over Excel format is its greater portability (CSV format is supported on all Xpress platforms) and the possibility to combine the *csv* driver freely it with other drivers, such as *shmem* or *rmt* that are not compatible with the other spreadsheet drivers.

3.2.4 Data output using *csv*

The following model `duosheet_out.mos` writes out the array `A` into the spreadsheet range F3:H3. These cells denote the first row of the rectangular area into which we wish to write.

```
model "Duo output (CSV)"
 uses "mmsheet"

 declarations
  A: array(-1..1,5..7) of real
 end-declarations

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

! Use an initializations block with the csv driver for writing data
 initializations to "mmsheet.csv:data.csv"
  A as "grow;[F3:H3]"
 end-initializations

end-model
```

In this model we have used the option `grow` of the *excel* driver to indicate that the actual output area may grow (= add more rows, the number of selected columns must be sufficient) beyond the specified output range as required by the data. Alternatively, we may also specify the complete output range such as in

```
 initializations to "mmsheet.csv:data.csv"
  A as '[F3:H11]'
 end-initializations
```

Yet another option that is available with the *csv* driver is the possibility to output the names of column headers via the option `skiph+` (note also that just like *xls/xlsx*, the *csv* driver will create the output file if it does not exist). The following version of our output example writes output into the columns F to H of a newly created file 'anewfile.csv' where the first row of each column contains the indicated header text (note that if the file and column headers exist already, the header text is expected to match the existing).

```
 initializations to "mmsheet.csv:anewfile.csv"
  A as 'skiph+;[F:H](Index1,Index2,Value_of_A)'
 end-initializations
```

3.2.5 Accessing simple CSV format files via *diskdata*

The *diskdata* I/O driver can be used to read or write simple CSV format files. This driver works with a single data table per file and whilst it is possible to make a selection among the columns (selecting columns from left to right only) the driver will always read all rows from the specified file. The *diskdata* driver operates in a line-wise fashion instead of loading the entire file into memory, which might be an advantage in situations where the available memory is limited. The following model `duodd.mos` implements the simple example we have seen in the previous sections with the *diskdata* I/O driver and equally using the `diskdata` subroutine that supports a more limited set of configuration options than the driver.

```
model "Duo input (diskdata)"
 uses "mmsheet"

 declarations
  A7,A8: dynamic array(range,range) of real
 end-declarations

! Use the diskdata driver for reading the data
 initializations from "mmetc.diskdata:"
  A7 as 'csv,datadd.csv'
 end-initializations

! Print out the data we have read
 writeln('A7 is: ', A7)

! Use the diskdata subroutine for reading the data
 diskdata(ETC_IN+ETC_CSV, 'datadd.csv', A8)
end-model
```

With the *diskdata* I/O driver it is possible to select columns by column numbers or by column headers when reading data:

```
initializations from "mmetc.diskdata:"
  A7 as 'csv(Index_i,Index_j,Value),skiph,datadd.csv'
  A8 as 'csv(1,2,3),datadd.csv'
 end-initializations
```

The 'diskdata' functionality can equally be used for writing out basic CSV-format files. With the I/O driver it is possible to configure a field separator or decimal separator character, but it does not support column selection of writing of header as this is the case for the spreadsheet drivers of *msheet*.

```
model "Duo output (diskdata)"
 uses "mmetc"

 declarations
  A: array(-1..1,5..7) of real
 end-declarations

 A :: [ 2,  4,  6,
```

```
       12, 14, 16,
       22, 24, 26]

! Using the diskdata driver
 initializations to "mmetc.diskdata:"
  A as 'csv,fsep=|;anewfile.csv'
 end-initializations

! Using the diskdata subroutine
 diskdata(ETC_OUT+ETC_CSV, 'anothernewfile.csv', A)
end-model
```

3.2.6 Working with spreadsheet ranges

In the preceding sections we have seen various forms of how to specify or refer to data ranges in a spreadsheet file. Table 2 provides an overview of the different possibilities supported by the I/O drivers of *mmsheet* and where applicable also the corresponding form that can be used with the *diskadata* driver from the module *mmetc*.

Table 2: Definition of spreadsheet ranges (where applicable with the corresponding formulation for the *diskdata* I/O driver)

		excel, xls/xlsx	***csv***	***diskdata***
Sheet reference				
by name	`'[Mysheet$]'`	yes	no	no
by counter	`'[2$]'`	yes	no	no
Named ranges	`'Myrange'`	yes	no	no
Explicit ranges				
entire (first) sheet	`'[]'`	yes	yes	`'csv,myfile.csv'`
column range	`'[B:D]'`	yes	yes	no
rectangular area	`'[B3:D6]'`	yes	yes	no
row-column notation	`'[R3C2:R6C4]'`	yes	yes	no
Column selection				
by column numbers	`'[A:Z](#2,#5,#9)'`	yes	yes	`'csv(2,5,9),myfile.csv'`
(unordered)	`'[A:Z](#5,#9,#2)'`	yes	yes	no
multiple occurrence	`'[A:Z](#2,#2,#9)'`	yes	yes	no
by column headers	`'skiph;[A:Z](Col1,Col3)'`	yes	yes	`'csv(Col1,Col3),skiph,myfile.csv'`
(unordered)	`'skiph;[A:Z](Col3,Col1)'`	yes	yes	no

The spreadsheet I/O drivers, including *csv*, allow users to retrieve size information about the defined ranges by specifying the option 'rangesize' (the forms marked with (*) in the code snippet below can be used with the *csv* driver)—note that the *excel* driver always returns all rows, not a row count for the populated area.

```
declarations
  l,l2,l3,l4,l5,l6,l7,l8,l9: list of integer   ! [height,width] of ranges
end-declarations

initialisations from 'mmsheet.xlsx:datarsz.xlsx'
  l as 'rangesize;[]'                  ! (*) Sizes for populated area in first sheet
  l2 as 'rangesize;[2$]'               ! Sizes for populated area in second sheet
  l3 as 'rangesize;[data$]'            ! Sizes for populated area in sheet 'data'
  l4 as 'rangesize;[a:c]'              ! (*) Sizes for populated area in columns A-C
  l5 as 'rangesize;[A:Z](#2,#7,#5)'    ! (*) Sizes for populated area in columns B-G
  l6 as 'rangesize;myrange'            ! Size of area defined by named range 'myrange'
  l7 as 'rangesize;myrange(#4,#2)'     ! Size of area between col.s 2-4 of 'myrange'
  l8 as 'rangesize;[R1C4:R9C8](#1,#2)' ! (*) Size of area from selected col.s in range
  l9 as 'rangesize;[D1:H9](#1,#2)'     ! (*) Size of area from selected col.s in range
end-initializations
```

```
writeln_("Data rows in spreadsheet range:", l(1), ", number of data columns:", l(2))
```

3.3 Oracle

When accessing Oracle databases using `initializations` blocks we need to use the OCI I/O driver name, `"mmoci.oci"`, followed by the database logon information, resulting in an extended file name such as `"mmoci.oci:myname/mypassword@dbname"`.

The introductory examples in this section are documented in full length. However, given the similarity of the ODBC and OCI interfaces, most of the examples in the 'Examples' section of this whitepaper are presented only in their ODBC version without repeating every time the modifications to the driver name/database connection string that are required to obtain their OCI version. Nevertheless, many examples are available with an Oracle version in the Examples Database on the Xpress website.

3.3.1 Data input using *oci*

Let us assume we are working with a database `dbname` that contains a table 'MyDataTable' with three fields, the first two (for the indices) of type `integer`, and the third of type `float`, filled with the data displayed in Section 3.1.1.

We may then use the following model `duooci.mos` to read in the array `A4` from the database and print it out. Only the module name in the `uses` statement and the extended filename for the database connection differ from what we have seen previously for an ODBC connection.

```
model "Duo input OCI (1)"
 uses "mmoci"

 declarations
  A4: dynamic array(range,range) of real
 end-declarations

! Use an initializations block with the odbc driver to read data
 initializations from "mmoci.oci:myname/mypassword@dbname"
  A4 as 'MyDataTable'
 end-initializations

! Print out the data we have read
 writeln('A4 is: ', A4)

end-model
```

3.3.2 Data output using *oci*

Outputting data from a Mosel model through the *oci* I/O driver again only requires few changes to the model version for ODBC. Consider the following example (file `duooci_out.mos`):

```
model "Duo output OCI (1)"
 uses "mmoci"

 declarations
  A: array(-1..1,5..7) of real
 end-declarations

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

! Use an initializations block with the oci driver for writing data
 initializations to "mmoci.oci:myname/mypassword@dbname"
```

```
  A as "MyOutTable1"
 end-initializations

end-model
```

The array `A` is written out to a data table named 'MyOutTable1' in the database `dbname`. This table must have been created before executing the Mosel model, with fields that correspond to the data array we want to write. That is, a total of three fields where the first two (index) fields have the type `integer` and the third field is of type `float`.

In terms of database functionality, when writing out data with `initializations to`, Mosel performs an "insert", no replacement or update. If the data table already contains some data, for instance from previous model runs, the new output will be appended to the existing data. This means that the insertion will fail if key fields have been defined and you are trying to write the same data entries a second time. The deletion of any existing data in the table(s) used for output must be done manually directly in the database prior to the model run, or by adding the corresponding SQL commands to the Mosel model. For the latter, it is possible to use the *oci* I/O driver in combination with other *mmoci* functionality, such as calling specific SQL queries (see Section 4.2).

4 Advanced example: using SQL queries

Certain tasks related to database access, such as deletion or update of existing data or the formulation of advanced selection statements, cannot be performed through `initializations` blocks. The ODBC and OCI modules therefore provide an alternative (lower level) means of accessing databases, namely using standard SQL commands.

4.1 ODBC

The module *mmodbc* defines the following subroutines for accessing external data sources through ODBC.

`SQLconnect, SQLdisconnect:`	Connect to a database / terminate the active connection.
`SQLexecute:`	Execute an SQL command.
`SQLreadinteger, SQLreadreal, SQLreadstring:`	Read an integer or real value, or a string from the database.
`SQLupdate`	Update the selected data with the provided array(s).

The procedures `SQLconnect`, `SQLdisconnect` and `SQLread...` can be used with any data source. `SQLupdate` only works if a data source supports positioned updates (this is typically the case for databases, but *not* for MS Excel). Depending on the data source the use of `SQLexecute` may be restricted to certain SQL commands: `select` and `insert` may be used with all data sources; commands like `create`, `delete`, and `update` will work with databases, but generally not with spreadsheets.

The procedures `SQLexecute` and `SQLupdate` allow the user to formulate his own SQL queries (using standard SQL). In this document we give a few examples of such queries, but these are by no means exhaustive. For a more thourough introduction to SQL the reader is refered to SQL tutorials and documentation such as those referenced at the site http://www.thefreecountry.com/documentation/onlinesql.shtml.

4.1.1 Data input with SQL statements

The following Mosel model corresponds to the model we have seen in Section 3.1.1 with the difference that we are now using SQL statements to read the data instead of an `initializations` block.

```
model "Duo input (2)"
 uses "mmodbc"

 declarations
  A5: dynamic array(range,range) of real
 end-declarations

! Use SQL statements to read the data
 SQLconnect('data.sqlite')
 if not getparam("SQLsuccess"): exit(1)
 SQLexecute("select Index_i,Index_j,Value from MyDataTable", A5)
 SQLdisconnect

! Print out the data we have read
 writeln('A5 is: ', A5)

end-model
```

The SQL statement `"select Index_i,Index_j,Value from MyDataTable"` says 'select fields `Index_i`, `Index_j`, and `Value` from a table called `MyDataTable`. If this table only contains these three fields and in the given order we might equally use the query `"select * from MyDataTable"` which says 'select everything from the range `MyDataTable`'. By using SQL statements directly in the Mosel model it is possible to have much more complex selection statements than the ones we have used here (see for instance the SQL queries formulated in Section 6.7).

If we wish to read data from the MS Access database `data.mdb`, we need to use the connection string `'DSN=MS Access Database;DBQ=C:/xpress/examples/data.mdb'` (or its short form `'data.mdb'`); similarly, for a MySQL database named `data` the corresponding string would be `'DSN=mysql;DB=data'`.

4.1.2 Data output with SQL statements

The example from Section 3.1.2 that outputs an array to a database via `initializations to` may be rewritten as follows with SQL statements:

```
model "Duo output (2)"
 uses "mmodbc"

 declarations
  A: array(-1..1,5..7) of real
 end-declarations

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

! Use SQL statements for writing data
 SQLconnect('data.sqlite')
 if not getparam("SQLsuccess"): exit(1)
 SQLexecute("insert into MyOutTable2 (Index1,Index2,AValue) values (?,?,?)", A)
 SQLdisconnect

end-model
```

The insertion command says 'write the contents of the array `A` in the form of value-triples to the fields

`Index1`, `Index2`, and `AValue` of the table `MyOutTable2`'. The question marks are placeholders for the index tuple, followed by the value of the array entry (first question mark = first index value, second question mark = second index value, ..., last question mark = array entry). Their number must correspond to the number of output table columns that are named. It is possible to select which indices/values to output and in which order (see Section 6.6). Please note that the third column of the output range has been given the header `AValue`: when writing data through SQL statements we cannot use the header `Value` since this is a reserved word for certain data sources. In the version of the example using `initializations to` the headers of the columns are not used (nevertheless, a header line must be present) and this word therefore does not cause any problems.

As explained for the first version of this example (Section 3.1.2) we need to make sure that the output range does not contain any data from previous runs by deleting data with the command sequence *Edit* ≫ *Delete* ≫ *Shift cells up* before the model execution.

For the Access database `data.mdb` the connection string would be `'DSN=MS Access Database;DBQ=C:/xpress/examples/data.mdb'` (or simply `'data.mdb'`). When writing to a database, we might remove data in the output table by hand, but it is certainly easier to clear the contents of this table by adding the following line to our Mosel model (immediately after the `SQLconnect` statement):

```
 SQLexecute("delete from MyOutTable2")
```

4.2 Oracle

The module *mmoci* defines the following subroutines for accessing Oracle databases via SQL commands.

`OCIlogon, OCIlogoff:`	Connect to a database / terminate the active connection.
`OCIexecute:`	Execute a PL/SQL command (select, insert, update, delete, create table, *etc.*).
`OCIreadinteger, OCIreadreal, OCIreadstring:`	Read an integer or real value, or a string from the database.
`OCIcommit, OCIrollback:`	Commit / roll back the current transaction (depending on setting of parameter `OCIautocommit`).

Please notice that there are some differences (other than the replacement of the prefix `SQL` by `OCI`) from the set of subroutines defined by module *mmodbc*: there is no separate 'update' procedure (data table updates can be formulated with `OCIexecute`), and an extra feature of this interface is the possibility to roll back transactions.

4.2.1 Data input with SQL statements

The following Mosel model corresponds to the model we have seen in Section 3.1.1 with the difference that we are now using SQL statements to read the data instead of an `initializations` block.

```
model "Duo input OCI (2)"
 uses "mmoci"

 declarations
  A5: dynamic array(range,range) of real
 end-declarations

! Use SQL statements to read the data
```

```
 OCIlogon("myname/mypassword@dbname")
 if not getparam("OCIsuccess"): exit(1)
 OCIexecute("select Index_i,Index_j,Value from MyDataTable", A5)
 OCIlogoff

! Print out the data we have read
 writeln('A5 is: ', A5)

end-model
```

An alternative, equivalent formulation of the database logon statement uses three separate strings for the user name, password, and database name:

```
 OCIlogon("myname", "mypassword", "dbname")
```

The SQL statement `"select Index_i,Index_j,Value from MyDataTable"` says 'select columns `Index_i`, `Index_j`, and `Value` from the table called `MyDataTable`. If this table contains only these three columns and in the given order we might equally use the query `"select * from MyDataTable"` which says 'select everything from the table `MyDataTable`'. In any case, we can work with exactly the same statement as in the ODBC version of this model since these are standard SQL queries. By using SQL statements directly in the Mosel model it is possible to have much more complex selection statements than the ones we have used here (see for instance the SQL queries formulated in Section 6.7).

4.2.2 Data output with SQL statements

The example from Section 3.1.2 that outputs an array to an Oracle database via `initializations to` may be rewritten as follows with SQL statements:

```
model "Duo output OCI (2)"
 uses "mmoci"

 declarations
  A: array(-1..1,5..7) of real
 end-declarations

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

! Use SQL statements for writing data
 OCIlogon('mmoci.oci:myname/mypassword@dbname')
 if not getparam("OCIsuccess"): exit(1)
 OCIexecute("delete from MyOutTable2")
 OCIexecute("insert into MyOutTable2 (Index1,Index2,AValue) values (:1,:2,:3)", A)
 OCIlogoff

end-model
```

The insertion command says 'write the contents of the array `A` in the form of value-triples to the fields `Index1`, `Index2`, and `AValue` of the table `MyOutTable2`'. The markers `:1`, `:2` *etc.* are placeholders for the index tuple, followed by the value of the array entry (`:1` = first index value, `:2` = second index value, `:3` = array entry). Their number must correspond to the number of output table columns that are named. By these markers it is possible to select which indices/values to output and in which order (see Section 6.6).

As explained for the I/O driver version of this example (Section 3.3.2) we need to make sure that the output table does not contain any data. Instead of removing data from the database table "by hand", this SQL implementation clears the contents of the output table by executing a 'delete' command in the

Mosel model before inserting the new data. Alternatively, we might have chosen to *update* the existing data in the output table by writing out array `A` with the following command:

```
OCIexecute("update MyOutTable2 set AValue=:3 where Index1=:1 and Index2=:2", A)
```

5 Some useful parameter settings

In this section we discuss some configuration options for the database and spreadsheet modules. For a complete list of the module parameters and I/O driver options the reader is refered to the module documentation in the corresponding chapters of the 'Mosel Language Reference Manual'.

5.1 Parameter settings to aid debugging

While developing an application that involves access to external data sources (and in particular when using ODBC) it is advisable to enable the output of error messages from the driver and possibly other debug information. The module *mmodbc* defines the following parameters to enable debug output and to retrieve information about the SQL statements that have been executed. To obtain the corresponding *mmoci* parameter names, replace the prefix `SQL` by `OCI`.

`SQLverbose`	Enable/disable mesasge printing by the ODBC driver.
`SQLdebug`	Enable/disable debug mode.
`SQLrowcnt`	Number of lines affected by the last SQL command.
`SQLrowxfr`	Number of lines transfered by the last SQL command.
`SQLsuccess`	Indicates whether the last SQL command succeeded.
`SQLconnection`	Identification number of the active connection to a database.

Parameters are set and retrieved with Mosel statements similar to the following:

```
setparam("SQLdebug", true)
writeln("Number of lines transfered: ", getparam("SQLrowxfr"))
```

It is recommended to check the execution status of individual SQL commands, and in particular of the connection to the database, via the parameter `SQLsuccess` and handle error situations suitably in the Mosel program, for example by raising an I/O error if database connection has failed:

```
SQLconnect("data.sqlite")
if not getparam("SQLsuccess"): setioerr("Database connection failed")
```

With the *odbc* driver you may also use the 'debug' option of the driver instead of the global setting:

```
initializations to "mmodbc.odbc:debug;data.xls"
 A as "MyOutTable1"
end-initializations
```

or

```
initializations to "mmodbc.odbc:data.xls"
 A as "debug;MyOutTable1"
end-initializations
```

In the first case, the setting applies to the whole `initializations` block, in the second case only to the specific statement (there may be any number of statements in a single block).

With the *oci* driver, a connection string including the 'debug' option as global setting will look as follows:

```
initializations to "mmoci.oci:debug;myusername/mypassword@dbname"
 A as "MyOutTable1"
end-initializations
```

5.2 Efficiency considerations

The *mmodbc* module fixes the maximum size of strings that are accepted to exchange data between Mosel and a database via the control `SQLcolsize`. This parameter is set by default to a relatively small value in order to avoid unnecessary overhead by reserving unused space—any portion of the strings exceeding the specified size is cut off. If a data set works with larger fields you need to increase the value of this setting, also aligning the setting of `SQLbufsize` that should always be larger then `SQLcolsize` (it is expected to be sufficiently large to store at least one entire row of data). To obtain the corresponding *mmoci* parameter names, replace the prefix `SQL` by `OCI`.

`SQLcolsize` Maximum length of strings accepted to exchange data.

`SQLbufsize` Size in kilobytes of the buffer used for exchanging data between Mosel and the ODBC driver.

The `bufsize` setting is also available as an option for the spreadsheet I/O drivers, increasing its default value might help to speed up the handling of very large data sets.

If a Mosel model employs a large number of write accesses to a database it is usually preferrably to enable *transaction mode*, provided that transactions are supported by the database (please refer to the documentation of your database; SQLite that comes with the *mmodbc* module supports this functionality, but it is not activated by default and needs to be explicitly enabled). This setting needs to be made before the database connection is opened. In transaction mode, database commands are collected up until a call to `SQLcommit` (commit changes) or `SQLrollback` (discard changes) is issued—with *mmoci* replace the prefix to subroutine and parameter names by `OCI`.

`SQLautocommit` If set to `false` enable transactions, otherwise any changes are sent immeditately to the database.

6 Examples

6.1 Outputting solution values

Writing the results of an optimization run to a database/spreadsheet is a two stage process.

1. Gather the solution data into a Mosel array.
2. Use ODBC/SQL/spreadsheet drivers to write to the external data source.

Using decision variables or constraints directly when writing out the data will *not* result in the solution values being written out.

The following Mosel model `soleg.mos` implements a tiny transportation problem using decision variables `x` of type `mpvar`. The array `SOL` receives the solution values of these variables. This array is then written out to an external data source (spreadsheet or database).

```
model "Solution values output"
 uses "mmxprs", "mmodbc"

 declarations
  R = 1..3
  S = 1..2
  SOL: array(R,S) of real        ! Array for solution values
  x: array(R,S) of mpvar         ! Decision variables
 end-declarations

! Define and solve the problem
 forall(i in R) sum(j in S) x(i,j) <= 4
 forall(j in S) sum(i in R) x(i,j) <= 6
 maximise( sum(i in R, j in S) (i*j)*x(i,j) )

! Get solution values from LP into the array SOL
 forall(i in R, j in S) SOL(i,j) := getsol(x(i,j))

! Use an initializations block with the odbc driver to write out data
 initializations to "mmodbc.odbc:soleg.sqlite"
  SOL as "MyOut1"
 end-initializations

end-model
```

The alternative method of using SQL statements for writing out the data looks as follows for this problem:

```
 SQLconnect("soleg.sqlite")
 if not getparam("SQLsuccess"): setioerr("Database connection failed")
 SQLexecute("insert into MyOut2 (First, Second, Solution) values (?,?,?)", SOL)
 SQLdisconnect
```

When working with a database, it may be preferable to use `SQLupdate` instead of `SQLexecute` to avoid having to clear the contents of the output table before every model run. (Notice that the 'update' command will only work if the table contains already data from previous runs). The corresponding SQL statement is:

```
 SQLupdate("select First, Second, Solution from MyOut2", SOL)
```

This command cannot be used with a spreadsheet; the results from previous runs always need to be removed by hand directly in the spreadsheet. With MS Excel we therefore recommend to use the *excel* driver instead of an ODBC connection to be able to overwrite any existing output data in the spreadsheet:

```
 initializations to "mmsheet.excel:soleg.xls"
  SOL as "skiph;MyOut1"
 end-initializations
```

Note: Instead of explicitly creating an array `SOL` to hold the solution values, it is also possible to create a temporary array immediately within the `initializations to` block using `evaluation of` in conjunction with the `array` operator:

```
 initializations to "mmodbc.odbc:soleg.sqlite"
  evaluation of array(i in R, j in S) x(i,j).sol as "MyOut1"
 end-initializations
```

Or the corresponding form within a SQL query:

```
 SQLexecute("insert into MyOut2 (First, Second, Solution) values (?,?,?)",
            array(i in R, j in S) x(i,j).sol)
```

6.2 Dense vs. sparse data format

All examples we have seen so far use *sparse* data format, *i.e.*, every data entry in the database tables or spreadsheets is given with its complete index tuple. If the index set(s) of an array are defined in the model or fixed otherwise it is also possible to work with data in *dense* format, *i.e.*, just the data entries without their index tuples.

6.2.1 Dense data format

The following example (file `indexeg.mos`) shows how data tables given in different formats may be read in by a Mosel model. We have enabled the 'debug' option to see the SQL statements generated by Mosel.

```
model ODBCImpEx
 uses "mmodbc"

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

 CSTR:= 'mmodbc.odbc:debug;indexeg.sqlite'

 ! Data must be dense - there are not enough columns to serve as index!
 initializations from CSTR
  A as 'Range3by2'
 end-initializations

 forall(i in 1..3)
  writeln("Row(",i,"): ", A(i,1), " ", A(i,2))

 ! Dense data
 initializations from CSTR
  B as 'noindex;Range2by3'
 end-initializations

 forall(i in 1..2)
  writeln("Row(",i,"): ", B(i,1), " ", B(i,2), " ", B(i,3))

 ! Indexed data
 initializations from CSTR
  C as 'Range2by3i'
 end-initializations

 forall(i in 1..2)
  writeln("Row(",i,"): ", C(i,1), " ", C(i,2), " ", C(i,3))

end-model
```

The first array, `A` is read from a table that helds data in dense format—just the data, no indices:

First	Second
1.2	2.2
2.1	2.2
3.1	4.4

This table has two columns and three rows. Since data in sparse format would require at least three columns, there is no confusion possible between the two formats and *mmodbc* will deduce automatically

the correct format to use. (However, if you wish to document clearly which data format is expected you may choose to add the option 'noindex' as in the following case.)

If an input table has a sufficiently large number of columns to serve as indices as is the case for the table `Range2by3` and array `B`, then the situation is ambiguous and we need to state explicitly that no indices are specified by using the option 'noindex'. Otherwise *mmodbc* will use its default (namely sparse format), which will lead to an error message in the present example since the data type (real) of the first two columns does not correspond to the type of the indices (integer) indicated in the model:

First	Second	Third
1.2	1.2	1.3
2.1	2.2	2.3

The third case, table `Range2by3i` and array `C`, corresponds to the (default) format that we have already seen in the previous examples—each data item is preceded by its index tuple. This table defines exactly the same data as the previous one:

Firsti	Secondi	Value
1	2	1.1
1	2	1.2
1	3	1.3
2	2	2.1
2	2	2.2
2	3	2.3

The same model as above may be rewritten with SQL commands instead of `initializations` blocks (the driver option 'noindex' is replaced by resetting the value of the *mmodbc* parameter `SQLndxcol`):

```
model ODBCImpEx2
 uses "mmodbc"

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

 CSTR:= 'indexeg.sqlite'

 SQLconnect(CSTR)
 if not getparam("SQLsuccess"): setioerr("Database connection failed")
 setparam("SQLdebug",true)

 ! Data must be dense - there are not enough columns to serve as index!
 SQLexecute("select * from Range3by2 ", A)
 forall(i in 1..3)
  writeln("Row(",i,"): ", A(i,1), " ", A(i,2))

 setparam("SQLndxcol", false)      ! Dense data
 SQLexecute("select * from Range2by3 ", B)
 forall(i in 1..2)
  writeln("Row(",i,"): ", B(i,1), " ", B(i,2), " ", B(i,3))

 setparam("SQLndxcol", true)       ! Indexed data
 SQLexecute("select * from Range2by3i ", C)
```

```
 forall(i in 1..2)
  writeln("Row(",i,"): ", C(i,1), " ", C(i,2), " ", C(i,3))

 SQLdisconnect

end-model
```

If using the *excel* driver instead of an ODBC connection, we need to use the `noindex` option also with the first data table ('Range3by2') since this driver does not do any 'guessing' about the table format.

```
model ODBCImpEx3
 uses "mmodbc"

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

 CSTR:= 'mmsheet.excel:indexeg.xls'

 ! Dense data ('noindex'), skipping the header line ('skiph')
 initializations from CSTR
  A as 'skiph;noindex;Range3by2'
 end-initializations

 forall(i in 1..3)
  writeln("Row(",i,"): ", A(i,1), " ", A(i,2))

 ! Dense data
 initializations from CSTR
  B as 'skiph;noindex;Range2by3'
 end-initializations

 forall(i in 1..2)
  writeln("Row(",i,"): ", B(i,1), " ", B(i,2), " ", B(i,3))

 ! Indexed data
 initializations from CSTR
  C as 'skiph;Range2by3i'
 end-initializations

 forall(i in 1..2)
  writeln("Row(",i,"): ", C(i,1), " ", C(i,2), " ", C(i,3))

end-model
```

6.2.2 Auto-indexation

A special case of dense data format are arrays that are indexed by a single index of type `range`. In this case it is possible to enable the *auto-indexation* feature when reading in data arrays from a data source that contains only the data values. The index value is generated via the row count while reading the data, starting from a specified index value or otherwise starting with the default value 1.
Auto-indexation applies to database access via *mmoci* and *mmodbc*, all spreadsheet drivers in the *mmsheet* module, and it is also available for the *diskdata* driver in *mmetc*. The following code snippets illustrate the use of this functionality (file `autoindex.mos`).

```
declarations
  A,A2,A3,B,C,B2,C2: array(range) of integer
end-declarations

initialisations from "mmsheet.xls:adt.xls" !"mmsheet.csv:adt.csv" "mmsheet.excel:adt.xls"
```

```
  A as 'autondx;[a:a]'              ! All data in column A
  A2 as 'autondx=-3;[a:a]'          ! Use start value -3 for indexation
  A3 as 'autondx;skiph;[](col1)'    ! Column with header 'col1'
  [B,C] as 'autondx=0;[a:b]'        ! Populating 2 arrays
  [B2,C2] as 'autondx;[b:b](#1,#1)' ! Selecting twice the same column
end-initialisations
```

With the *diskdata* I/O driver the start value can be specified in the same way as for the spreadsheet drivers, the `diskdata` subroutine only works with the default start value.

```
initialisations from 'mmetc.diskdata:'
  A as 'csv(1),autondx;adt.csv'      ! Selecting the first column
  A2 as 'csv(1),autondx=-3;adt.csv'  ! Use start value -3 for indexation
  [B,C] as 'csv,autondx;adt.csv'     ! Populating 2 arrays
end-initialisations
diskdata(ETC_IN+ETC_CSV+ETC_AUTONDX,'adt.csv',[B2,C2])
```

For databases, the auto-indexation is enabled via the control parameter `SQLautondx`, and a start value different from the default value 1 can be specified by setting `SQLfirstndx`.

```
SQLconnect("adt.sqlite")
if not getparam("SQLsuccess"): setioerr("Database connection failed")
setparam("SQLautondx",true);                ! Enable auto-indexation
SQLexecute("select (Col1) from MyTable", A)  ! Selecting a single column
setparam("SQLfirstndx",-3)                  ! Use start value -3 for indexation
SQLexecute("select (Col1) from MyTable", A2)
setparam("SQLfirstndx",1)                   ! Revert to default start value
SQLexecute("select (Col1,Col2) from MyTable", [B,C])   ! Populating 2 arrays
SQLdisconnect
```

6.2.3 Multidimensional tables in rectangular format

Quite frequently, particularly when working with spreadsheets, multidimensional input data arrays are formatted in 2-dimensional (rectangular) form. For example, take a look at the following table: we want to populate an array A_{ijk} with 3 dimensions from the data held in seven columns, where the first 2 columns are indices (i and j) and the remaining columns are data values corresponding to different index values for the third and last index k of the array.

Firsti	Secondi	Value_1	Value_2	Value_3	Value_4	Value_5
2	B	22.1	22.2	22.3	22.4	22.5
1	D	14.1	14.2	14.3	14.4	14.5

This table can be read by the following Mosel code (see example file `threedimarr.mos`). Notice that the contents of the first two index sets is not defined in the model—their contents is read with the input data. However, the last index k that is written 'across the table columns' needs to be defined in the model and it has to be of type `range` (that is, using a set type that has an ordering) to make sure that the entries will be read in the same order as expected by the model. With this data format we need to use the `partndx` option of the I/O drivers to indicate that not all indices are to be read in with the input data. Furthermore, as the input data only defines values for a few index tuples, we use a *dynamic* array that will only contain those elements for which values are specified in the data file.

```
model "ThreeDimArr"
 uses "mmodbc"

 declarations
  I: range
  J: set of string
```

```
 K = 1..5               ! The last index set must be defined in the model
 A: dynamic array(I,J,K) of real
end-declarations

initialization from 'mmsheet.excel:partndx;threedim.xls'
 A as 'Tab_23'
end-initializations

writeln("A: ")
forall(i in I, j in J, k in K | exists(A(i,j,k)))
 writeln("A(", i, ",", j, ",", k, "): ", A(i,j,k))

end-model
```

To read the same data with the *odbc* driver we can use

```
initialization from 'mmodbc.odbc:partndx;threedim.sqlite'
 A as 'Tab_23'
end-initializations
```

and to obtain the corresponding version using SQL statements, the `initializations` block can be replaced by the following lines.

```
declarations
 Idx3: text
end-declarations

SQLconnect("threedim.sqlite")
if not getparam("SQLsuccess"): exit(1)

setparam("SQLndxcol", false)   ! Partially indexed data

forall(k in K) Idx3+= (", Value_"+k )
SQLexecute("select Firsti, Secondi" + Idx3 + " from Tab_23", A)
SQLdisconnect
```

6.3 Reading several arrays from a single table

If two or more data arrays have the same index sets, then their values may be defined in a single spreadsheet range/database table, such as the following table where the first two columns hold the indices and the last two colums the data entries for two arrays:

Products	Mach	Cost	Duration
prod1	1	1.2	3
prod1	3	2.4	2
prod2	3	3	1
prod2	2		2
prod4	1	4	5
prod4	4	3.2	2
prod3	3	5.7	2
prod3	4	2.9	8
prod3	1	3	

Notice that in this table not all entries are defined for every array.

The following Mosel model `multicol.mos` reads the data from the range `ProdData` into two array,

COST and DUR. For every array only those entries that are specified in the input data will actually be defined:

```
model "Multiple data columns"
 uses "mmodbc"

 declarations
  PRODUCTS: set of string
  MACH: range
  COST: dynamic array(PRODUCTS,MACH) of real
  DUR: dynamic array(PRODUCTS,MACH) of integer
 end-declarations

 initializations from "mmodbc.odbc:multicol.sqlite"
  [COST,DUR] as 'ProdData'
 end-initializations

 writeln(COST); writeln(DUR)

end-model
```

The SQL version of this model is as follows:

```
 SQLconnect('multicol.sqlite')
 if not getparam("SQLsuccess"): exit(1)
 setparam("SQLverbose",true)
 SQLexecute("select * from ProdData ", [COST,DUR])
 SQLdisconnect
```

If we wish to read data from a different database, also defining the table ProdData, we again simply need to adapt the filename or the connection string to the database name.

To use the *excel* driver with a data range definition that includes the header line the option skiph needs to be employed.

```
 initializations from "mmsheet.excel:multicol.xls"
  [COST,DUR] as 'skiph;ProdData'
 end-initializations
```

6.4 Outputting several arrays into a single table

Similarly to what we have seen in the previous section for data input we may also write out several data arrays to a single spreadsheet range/database table, provided that all arrays have the same index sets.

The following example multiout.mos reads in two data arrays from a text file in Mosel format and outputs them to a spreadsheet range and also to a database table. Both the spreadsheet range and the database table must have been created before this model is run.

```
model "Output multiple data columns"
 uses "mmodbc"

 declarations
  PRODUCTS: set of string
  MACH: range
  COST: dynamic array(PRODUCTS,MACH) of real
  DUR: dynamic array(PRODUCTS,MACH) of integer
 end-declarations

! Read data
 initializations from "multiout.dat"
  COST DUR
 end-initializations
```

```
! Write data to the Access database multicol.mdb
! (this assumes that the table 'CombData' has been created previously):
 initializations to "mmodbc.odbc:debug;multicol.mdb"
  [COST,DUR] as 'CombData'
 end-initializations

! Write data to the Excel spreadsheet multicol.xls
! (this assumes that the range 'CombData' has been created previously):
 initializations to "mmsheet.excel:skiph;grow;multicol.xls"
  [COST,DUR] as 'CombData'
 end-initializations

! Alternative: specify the range/worksheet
 initializations to "mmsheet.excel:multicol.xls"
  [COST,DUR] as 'grow;[Sheet1$L4:O4]'
 end-initializations

end-model
```

The input data file contains the same data as has been used for the previous example, that is, different entries are defined for every array. In the resulting output tables some entries will therefore be left empty.

```
COST: [(prod1 1) 1.2 (prod1 3) 2.4 (prod2 3) 3 (prod4 1) 4 (prod4 4) 3.2
       (prod3 3) 5.7 (prod3 4) 2.9 (prod3 1) 3  ]

DUR: [(prod1 1) 3 (prod1 3) 2 (prod2 3) 1 (prod2 2) 2 (prod4 1) 5
      (prod4 4) 2 (prod3 3) 2 (prod3 4) 8 ]
```

The SQL version of the model above may look as follows. For the database, we have added an SQL command ('create') that creates the database table before the data is written out. This command cannot be used with Excel spreadsheets: the output range must be prepared before the Mosel model is run. In this example the fields of the database table/columns of the spreadsheet range use the same names as our Mosel model entities: this is just coincidence and by no means a necessity.

```
! Write data to the Access database multicol.mdb
! (create the output table and then output the data)
 SQLconnect('multicol.mdb')
 if not getparam("SQLsuccess"): setioerr("Database connection failed")
 setparam("SQLdebug", true)
 SQLexecute("create table CombData (Products varchar(10), Mach integer, Cost double,
 Duration integer)")
 SQLexecute("insert into CombData(Products, Mach, Cost, Duration) values (?,?,?,?)",
            [COST,DUR])
 SQLdisconnect
```

In this example the insertion statements output quadruples (denoted by the four question marks), each consisting of an index tuple, followed by the corresponding entries of the two arrays in the given order. That is, an output tuple has the form (i,j,COST(i,j),DUR(i,j)). These tuples are written into the four selected columns of the table `CombData`.

6.5 Reading an array from several tables

Especially when working with arrays of more than two dimensions it may happen that the input data is split into several spreadsheet ranges/database tables.

We wish to read an array, `INCOME`, indexed by the sets `CUST` and `PERIOD` from three tables (one table per customer). In the first instance, assume that every data table includes both the `CUST` and the

PERIOD index column, such as (table COLDAT1):

CUST	PERIOD	INCOME
1	1	11
1	2	21
1	3	31
1	4	41
1	5	51

In this case we may read in the data with three statements within a single initializations block:

```
model "multiple data sources"
 uses "mmodbc"

 declarations
  CUST: set of integer
  PERIOD: range
  INCOME: dynamic array(CUST,PERIOD) of real
 end-declarations

! Method 1: Data in columns, with CUST index value included
 initializations from 'mmodbc.odbc:multitab.sqlite'
  INCOME as 'COLDAT1'
  INCOME as 'COLDAT2'
  INCOME as 'COLDAT3'
 end-initializations

 writeln("1: ", INCOME)

end-model
```

The same with SQL statements:

```
 SQLconnect('multitab.sqlite')
 if not getparam("SQLsuccess"): setioerr("Database connection failed")
 SQLexecute("select CUST,PERIOD,INCOME from COLDAT1", INCOME)
 SQLexecute("select CUST,PERIOD,INCOME from COLDAT2", INCOME)
 SQLexecute("select CUST,PERIOD,INCOME from COLDAT3", INCOME)
 SQLdisconnect
```

Now assume that the input data table for each customer only contains the PERIOD index and the data value itself:

PERIOD	INCOME
1	11
2	21
3	31
4	41
5	51

In this case we need to introduce an auxiliary array TEMP with just one index into which we read the data for every customer and that gets copied into the array INCOME. (This method supposes that we know the contents of the set CUST; with the first method this was not required.)

```
! Method 2: Data in columns, without CUST index value
 procedure readcol(cust:integer, table:string)
  declarations
   TEMP: array(PERIOD) of real
  end-declarations

  initializations from 'mmodbc.odbc:multitab.sqlite'
   TEMP as table
  end-initializations
  forall(p in PERIOD) INCOME2(cust,p):=TEMP(p)
 end-procedure

 forall(c in CUST) readcol(c, "COLDAT"+c+"A")
 writeln("2: ", INCOME)
```

If we wish to employ SQL statements for reading the data, the procedure `readcol` may look as follows (all else remains unchanged):

```
 procedure readcol(cust:integer, table:string)
  declarations
   TEMP: array(PERIOD) of real
  end-declarations

  SQLexecute("select PERIOD,INCOME from "+table, TEMP)
  forall(p in PERIOD) INCOME(cust,p):=TEMP(p)
 end-procedure
```

In this case and with the rowwise representation shown below the formulation with SQL statements is likely to be more efficient since we only need to connect once to the data source and then execute a series of 'select' commands. For `initializations` blocks we open and close an ODBC connection with every new block, that is, at every execution of the procedure `readcol`.

As a third case consider a representation of the data in transposed form, that is, not columnwise but rowwise as shown in the following example table. (This format may occur with spreadsheets but it is certainly less likely, though not impossible, with databases.) With Excel we always need to define a header row for a data range—here we have simply filled it with zeros since its contents is irrelevant. Any row headers written at the left or right of the data range are purely informative, they must not be selected as part of the range.

DUMMY	0	0	0	0	0	
PERIOD	1	2	3	4	5	
INCOME	11	21	31	41	51	

Such rowwise formatted data may be read with the following Mosel code. As with the previous method, we define a procedure `readrow` to read data from a single data range. Both index sets, `CUST` and `PERIOD`, must be known and the set `PERIOD` must be finalized (this means that its contents cannot change any more and the set is treated by Mosel similarly to a constant set).

```
! Method 3: Data in rows, without CUST index value
 procedure readrow(cust:integer, table:string)
  declarations
   TEMP: array(1..2,PERIOD) of real
  end-declarations

  initializations from 'mmsheet.excel:multitab.xls'
   TEMP as 'noindex;'+table
```

```
 end-initializations
 forall(p in PERIOD) INCOME3(cust,p):=TEMP(2,p)
end-procedure

finalize(PERIOD)                        ! The index sets must be known+fixed
forall(c in CUST) readrow(c, "ROWDAT"+c)
writeln("3: ", INCOME3)
```

The corresponding SQL code looks as follows (notice the setting of the parameter `SQLndxcol`):

```
procedure readrow(cust:integer, table:string)
 declarations
  TEMP: array(1..2,PERIOD) of real
 end-declarations

 SQLexecute("select * from "+table, TEMP)
 forall(p in PERIOD) INCOME3(cust,p):=TEMP(2,p)
end-procedure

finalize(PERIOD)                ! The index sets must be known+fixed
setparam("sqlndxcol",false)     ! Data specified in dense format (no indices)
forall(c in CUST) readrow(c, "ROWDAT"+c)
setparam("sqlndxcol",true)
```

To read the data from a database using the *odbc* driver instead of *excel* we merely need to change the file name:

```
initializations from 'mmodbc.odbc:skiph;multitab.sqlite'
 ...
```

6.6 Selection of columns/fields

The structure of the tables read from or written to using ODBC does not necessarily have to be the same as the tables in the Mosel model: tables may have more fields than required, or fields may be defined in a different order. To choose the fields from such tables that we wish to access we need to indicate the field names in the ODBC queries. In some of the previous SQL examples we have already named the fields we wish to access (instead of using a wildcard, such as `select * from`). With `initializations` blocks it is equally possible to indicate the names of the fields as is shown in the following example.

We work with the example from Sections 6.3 and 6.4 where a single table in the data source holds data for several Mosel arrays. The following Mosel model `odbcinv.mos` reads in the two arrays `COST` and `DUR` separately. The index sets of the array `COST` are in inverse order.

```
model "ODBC selection of columns"
 uses "mmodbc"

 declarations
  PRODUCTS: set of string
  MACH: range
  COST: dynamic array(MACH,PRODUCTS) of real
  DUR: dynamic array(PRODUCTS,MACH) of integer
 end-declarations

 initializations from "mmodbc.odbc:debug;multicol.mdb"
  COST as "ProdData(Mach,Products,Cost)"
  DUR as "ProdData(Products,Mach,Duration)"
 end-initializations

! Print out what we have read
 writeln(COST); writeln(DUR)
```

```
! Delete and re-create the output table
 SQLconnect('multicol.mdb')
 if not getparam("SQLsuccess"): setioerr("Database connection failed")
 SQLexecute("drop table CombData2")
 SQLexecute("create table CombData2 (Products varchar(10), Mach integer, Cost double,
 Duration integer)")
 SQLdisconnect

 initializations to "mmodbc.odbc:debug;multicol.mdb"
  COST as "CombData2(Mach,Products,Cost)"
  DUR as "CombData2(Products,Mach,Duration)"
 end-initializations

end-model
```

When writing out the two arrays into the result table `CombData` using `initializations to` the data does not appear the way we would wish: the data for the second array gets appended to the data of the first instead of filling the remaining field with the additional data. The reason for this is that `initializations to` performs an 'insert' command and not an 'update' which is the command to use if the table already holds some data. To fill the table in the desired way it is therefore necessary to use SQL queries for completing the output. Below follows the complete SQL version of this model.

```
 declarations
  TEMP: array(PRODUCTS,MACH) of integer
 end-declarations

 setparam("SQLdebug",true)
 SQLconnect('multicol.mdb')
 if not getparam("SQLsuccess"): setioerr("Database connection failed")

! Read data from the table 'ProdData'
 SQLexecute("select Mach,Products,Cost from ProdData", COST)
 SQLexecute("select Products,Mach,Duration from ProdData", DUR)

! Print out what we have read
 writeln(COST); writeln(DUR)

! Write out data to another table (after deleting and re-creating the table)
 SQLexecute("drop table CombData")
 SQLexecute("create table CombData (Products varchar(10), Mach integer, Cost double,
 Duration integer)")

! Write out the 'COST' array
 SQLexecute("insert into CombData (Mach,Products,Cost) values (?,?,?)", COST)

! Fill the 'Duration' field of the output table:
! 1. update the existing entries, 2. add new entries
 SQLupdate("select Products,Mach,Duration from CombData", DUR)
 forall(p in PRODUCTS, m in MACH | exists(DUR(p,m)) and not exists(COST(m,p)))
  TEMP(p,m) := DUR(p,m)
 SQLexecute("insert into CombData (Products,Mach,Duration) values (?,?,?)",
            TEMP)

 SQLdisconnect
```

A second possibility for formulating the SQL output query for the array `COST` is to use the numbering of columns (`?1`, `?2`, *etc.*) to select which of the indices/value columns of the data array we want to write out (we might choose, for instance, to write out only a single index set), and in which order. This functionality has no direct correspondence in the formulation with `initializations to` blocks.

```
 SQLexecute("insert into CombData (Products,Mach,Cost) values (?2,?1,?3)",
            COST)
```

There is also an equivalent formulation of the 'update' statement using the SQL command 'update' instead of `SQLupdate`. We use again the numbering of columns to indicate where the indices and data entries of the Mosel array `DUR` are to be inserted:

```
SQLexecute("update CombData set Duration=?3 where Products=?1 and Mach=?2", DUR)
```

6.7 SQL selection statements

As has been said before, using SQL statements instead of `initializations` blocks gives the user considerably more freedom in the formulation of his SQL queries. In this section we are going to show examples of advanced functionality that cannot be achieved with `initializations` blocks.

We are given a database with two tables. The first table, called `MYDATA`, has the following contents.

ITEM	COST	DIST
A	10	100
B	20	2000
C	30	300
D	40	5000
E	50	1659

The second table, `USER_OPTIONS`, defines a few parameters, that is, it has only a single entry per field/column. We may perform, for instance, the following tasks:

- Select all data entries from table `MYDATA` for which the `DIST` value is greater than the value of the parameter `MINDIST` in the table `USER_OPTIONS`.
- Select all data entries with indices among a given set.
- Select all data entries for which the ratio `COST/DIST` lies within a given range.
- Retrieve the data entry for a given index value.
- Apply some functions to the database entries.

The following Mosel model `odbcselfunc.mos` shows how to implement these tasks as SQL qeries.

```
model "ODBC selection and functions"
 uses "mmodbc"

 declarations
  Item: set of string
  COST1,COST2,COST3: dynamic array(Item) of real
 end-declarations

 setparam("SQLdebug",true)
 SQLconnect('odbcsel.mdb')
 if not getparam("SQLsuccess"): setioerr("Database connection failed")

! Select data depending on the value of a second field, the limit for which
! is given in a second table USER_OPTIONS
 SQLexecute("select ITEM,COST from MYDATA where DIST > (select MINDIST from
 USER_OPTIONS)", COST1)

! Select data depending on the values of ITEM
 SQLexecute("select ITEM,COST from MYDATA where ITEM in ('A', 'C', 'D', 'G')",
```

```
            COST2)

! Select data depending on the values of the ratio COST/DIST
 SQLexecute("select ITEM,COST from MYDATA where COST/DIST between 0.01 and 0.1",
            COST3)

 writeln(COST1, COST2, COST3)

! Print the DIST value of item 'B'
 writeln(SQLreadreal("select DIST from MYDATA where ITEM='B'"))

! Number of entries with COST>30
 writeln("Count COST>30: ",
         SQLreadinteger("select count(*) from MYDATA where COST>30"))

! Total and average distances
 writeln("Total distance: ", SQLreadreal("select sum(DIST) from MYDATA"),
         ", average distance: ", SQLreadreal("select avg(DIST) from MYDATA"))

end-model
```

6.8 Accessing structural information from databases

With SQL commands, it is possible to access detailed information about the contents of a database, including the complete list of tables and for each table, the names and types of its fields. The model `odbcinspectdb.mos` printed below shows how to retrieve and display the structural information for a given database.

```
model "Analyze DB structure"
 uses "mmodbc"

 declarations
  tables: list of string
  pkeylist: list of string
  pkeyind: list of integer
  fnames: dynamic array(Fields: range) of string
  ftypes: dynamic array(Fields) of integer
  ftypenames: dynamic array(Fields) of string
 end-declarations

 setparam("SQLverbose",true)
 SQLconnect("personnel.sqlite")
 if not getparam("SQLsuccess"): setioerr("Database connection failed")

! Retrieve list of database tables
 SQLtables(tables)
 forall(t in tables) do

  ! Retrieve primary keys
   SQLprimarykeys(t, pkeylist)
   writeln(t, " primary key field names: ", pkeylist)
   SQLprimarykeys(t, pkeyind)
   writeln(t, " primary key field indices: ", pkeyind)

  ! Retrieve table structure
   writeln(t, " has ", SQLcolumns(t,fnames,ftypes), " fields")
   res:=SQLcolumns(t,fnames,ftypenames)
   forall(f in Fields | exists(fnames(f)))
     writeln(f, ": ", fnames(f), " ", ftypes(f), ": ", ftypenames(f))

  ! Delete aux. arrays for next loop iteration
   delcell(fnames); delcell(ftypes); delcell(ftypenames)
 end-do
```

```
 SQLdisconnect
end-model
```

With the spreadsheet I/O drivers of *mmsheet* it is possible to retrieve size information for ranges: see the description of option 'rangesize' in Section 3.2.6.

6.9 Working with lists

Data in spreadsheets or databases is stored in the form of ranges or tables and so far we have always used Mosel arrays as the corresponding structure within our models. Yet there are other possibilities. In this section we shall see how to work with Mosel lists in correspondence to 1-dimensional tables/ranges in the data source. The next section shows how to work with the Mosel data structure 'record'.

Assume we are given a spreadsheet `listdata.xls` with two 1-dimensional ranges, `List1` and `List2` and an integer `A`:

	List1								
	1	2	3	4	5	6	7	8	
	A			***List2***					
	2002			Jan	May	Jul	Nov	Dec	

The following Mosel model `listinout.mos` reads in the two ranges as lists and also the integer `A`, makes some modifications to each list and writes them out into predefined output ranges in the spreadsheet.

```
model "List handling (Excel)"
 uses "mmsheet"

 declarations
  R: range
  LI: list of integer
  A: integer
  LS,LS2: list of string
 end-declarations

 initializations from "mmsheet.excel:listdata.xls"
  LI as "List1"
  A
  LS as "List2"
 end-initializations

! Display the lists
 writeln("LI: ", LI)
 writeln("A: ", A, ", LS: ", LS)

! Reverse the list LI
 reverse(LI)

! Append some text to every entry of LS
 LS2:= sum(l in LS) [l+" "+A]

! Display the modified lists
 writeln("LI: ", LI)
 writeln("LS2: ", LS2)
```

```
 initializations to "mmsheet.excel:listdata.xls"
  LI as "List1Out"
  LS2 as "List2Out"
 end-initializations

end-model
```

Please note that we may choose as input ranges a column or a row of a spreadsheet. Similarly, when using the *excel* driver to access the spreadsheet the output area of a list may also be a column or a row. List data in databases is always represented as a field of a database table.

The same model implemented with SQL commands looks as follows.

```
 setparam("SQLverbose",true)
 SQLconnect("listdata.sqlite")
 if not getparam("SQLsuccess"): setioerr("Database connection failed")
 SQLexecute("select * from List1", LI)
 A:= SQLreadinteger("select * from A")
 SQLexecute("select * from List2", LS)

...

 SQLexecute("delete from List1Out")     ! Cleaning up previous results: works
 SQLexecute("delete from List2Out")     ! only for databases, cannot be used
                                        ! with spreadsheets (instead, delete
                                        ! previous solutions directly in the
                                        ! spreadsheet file)
 SQLexecute("insert into List1Out values (?)", LI)
 SQLexecute("insert into List2Out values (?)", LS2)
 SQLdisconnect
```

The modules *mmodbc*, *mmoci* and *mmsheet* do not accept composed structures involving lists like 'array of list' (such constructs are permissible when working with text files in Mosel format).

6.10 Working with records

In this section we work once more with the data range `ProdData` that has already been used in the example of Section 6.3:

Products	Mach	Cost	Duration
prod1	1	1.2	3
prod1	3	2.4	2
prod2	3	3	1
prod2	2		2
prod4	1	4	5
prod4	4	3.2	2
prod3	3	5.7	2
prod3	4	2.9	8
prod3	1	3	

We now want to read this data into a record data structure, more precisely, an array of records where each record contains the data for one product-machine pair. Such a record may be defined in different ways: it may contain just the fields 'Cost' and 'Duration', using the product and machine as indices, or we could define a record with four fields, 'Product', 'Mach', 'Cost', and 'Duration', using a simple counter as index to the array. The model `recordin.mos` printed below implements both cases.

```
model "Record input (Excel)"
 uses "mmsheet"

 declarations
  PRODUCTS: set of string
  MACH: range
  ProdRec = record
   Cost: real
   Duration: integer
  end-record
  PDATA: dynamic array(PRODUCTS,MACH) of ProdRec

  R = 1..9
  AllDataRec = record
   Product: string
   Mach: integer
   Cost: real
   Duration: integer
  end-record
  ALLDATA: array(R) of AllDataRec
 end-declarations

! **** Reading complete records

 initializations from "mmsheet.excel:recorddata.xls"
  PDATA as "ProdData"
  ALLDATA as "noindex;ProdData"
 end-initializations

! Now let us see what we have
 writeln('PDATA is: ', PDATA)
 writeln('ALLDATA is: ', ALLDATA)

end-model
```

This model will fill the fields of each record in the order of their definition with the data from a row of the input range in the order of the columns. That is, the first two columns of range `ProdData` will become the indices of `PDATA`, the third column is read into the 'Cost' field, and the forth column into the 'Duration' field. The record array `ALLDATA` will have the first column of `ProdData` in its first field ('Product'), the second column in the field 'Mach', and so on.

It is also possible (a) to select certain columns from a database table or spreadsheet range and (b) to specify which record fields to initialize. The former can be used to read data from a spreadsheet range or database table that contains other data or has columns/fields arranged in a different order from the Mosel model as we have already seen in the example of Section 6.6. The following code extract shows how to read the contents of some record fields from specified parts of the input data range.

```
 declarations
  PDATA2: dynamic array(PRODUCTS,MACH) of ProdRec
  ALLDATA2: array(R) of AllDataRec
 end-declarations

! **** Reading record fields

 initializations from "mmodbc.odbc:recorddata.sqlite"
  PDATA2(Cost) as "ProdData(IndexP,IndexM,Cost)"
  ALLDATA2(Product,Mach,Duration) as "noindex;ProdData(IndexP,IndexM,Duration)"
 end-initializations
```

This results in an array of records `PDATA2` with values in the 'Cost' field and all 'Duration' fields at 0 and an array of records `ALLDATA2` with values in the 'Product', 'Mach', and 'Duration' fields and all 'Cost' fields at 0.

When using the *excel* driver for accessing a spreadsheet it is equally possible to select columns from a spreadsheet range, either via column header names if they are included in the range specification or via column order numbers within the seleted range:

```
initializations from "mmsheet.excel:recorddata.xls"
 PDATA as "skiph;ProdData"
 ALLDATA as "skiph,noindex;AllData"
 PDATA2(Cost) as "ProdData(#1,#2,#3)"
 ALLDATA2(Product,Mach,Duration) as "noindex;ProdData(#1,#2,#4)"
end-initializations
```

With SQL statements it would be possible to select columns from a spreadsheet range (or database table fields). However Mosel's syntax does not provide any means to select fields for an array of records in the `SQLexecute` statement. We can initialize the complete arrays of records as shown below, but it is not possible to select just certain record fields when reading or writing data (it would of course be possible to employ some auxiliary data structures for reading in the data and copy their contents to the array of records).

```
setparam("SQLverbose",true)
SQLconnect("recorddata.xls")
if not getparam("SQLsuccess"): exit(1)
SQLexecute("select * from ProdData", PDATA)
setparam("SQLndxcol", false)           ! Dense data
SQLexecute("select * from AllData", ALLDATA)
SQLdisconnect
```

Note further that any (record) types used in SQL statements must be declared as `public`.

```
public declarations
 ProdRec = public record
  !...
 end-record
 AllDataRec = public record
  !...
 end-record
end-declarations
```

6.11 Handling dates and time

Fields of databases that are defined as date or time types find their direct correspondence in the types `date`, `time`, or `datetime` of the Mosel module *mmsystem*. The modules *mmodbc*, *mmoci* and *mmsheet* support these types for reading and writing data and we explain here how to work with them.

Dates and times are passed in their textual representation from a database to Mosel (or from Mosel to the database). The representation of date and time information within databases is different from one product to another and may not be compatible with Mosel's default format. The first step when starting to work with date and time related data therefore always is to retrieve sample data in the form of a string and print it out to analyze its format. This can be done by a few lines of Mosel code, such as:

```
declarations
 sd,st: string
end-declarations

initializations from "datetest.dat"
 sd as "ADate"
 st as "ATime"
end-initializations

writeln("sd: ", sd, ", st: ", st)
```

The date and time formats are defined by setting the parameters `timefmt`, `datefmt`, and `datetimefmt` of module *mmsystem*. The encoding of the format strings is documented in the 'Mosel Language Reference Manual', Chapter 'mmsystem'.

In the model displayed below we read a first set of dates/times that are defined as such in the data source. The second set are simply strings in the data source and Mosel transforms them into dates/times according to the format defined by our model before reading the data. For the output we use Mosel's own format; depending on the data source the result will be interpreted as strings or as time/date data.

```
model "Dates and times (ODBC)"
 uses "mmsystem", "mmodbc"

 declarations
  T: time
  D: date
  DT: datetime
  Dates: list of date
 end-declarations

! Select the format used by the spreadsheet/database
! (database fields have date/time types)
 setparam("timefmt", "%y-%0m-%0d %0H:%0M:%0S")
 setparam("datefmt", "%y-%0m-%0d")
 setparam("datetimefmt", "%y-%0m-%0d %0H:%0M:%0S")

 initializations from "mmodbc.odbc:datetime.mdb"
  T as "Time1"
  D as "Date1"
  DT as "DateTime1"
  Dates as "Dates"
 end-initializations

 setparam("timefmt", "%h:%0M %p")
 writeln(D, ", ", T)
 writeln(DT)
 writeln(Dates)

! Read date / time from strings (database fields have some string type)
 setparam("timefmt", "%Hh%0Mm")
 setparam("datefmt", "%dm%0my%0y")
 setparam("datetimefmt", "%dm%0my%0y, %Hh%0Mm")

 initializations from "mmodbc.odbc:datetime.mdb"
  T as "Time2"
  D as "Date2"
  DT as "DateTime2"
 end-initializations

 writeln(D, ", ", T)
 writeln(DT)

! Use Mosel's default format
 setparam("timefmt", "")
 setparam("datefmt", "")
 setparam("datetimefmt", "")

 writeln(D, ", ", T)
 writeln(DT)

! The following assumes that the database output fields have type string
! since we are not using the date/time formatting expected by the database
 initializations to "mmodbc.odbc:datetime.mdb"
  T as "TimeOut"
  D as "DateOut"
  DT as "DateTimeOut"
 end-initializations
```

```
end-model
```

The formatting for dates and times at the beginning of the model where we read database fields with date/time types (`Time1`, `Date1`, `DateTime1`, and `Dates`) applies to Access and Excel read through ODBC. For an SQLite or mysql database this would be

```
setparam("timefmt", "%0H:%0M:%0S")
setparam("datefmt", "%y-%0m-%0d")
setparam("datetimefmt", "%y-%0m-%0d %0H:%0M:%0S")
```

and an Oracle database uses the following format:

```
setparam("timefmt", "%0d-%N-%0Y %0h.%0M.%0S.%0s %P")
setparam("datefmt", "%0d-%N-%0Y")
setparam("datetimefmt", "%0d-%N-%0Y %0h.%0M.%0S.%0s %P")
```

The *xls* and *xlsx* drivers receive dates, times and timestamps in encoded form, the conversion to the text form always uses the default format of *mmsystem*. This means that we can simply leave out the `setparam` calls at the beginning of the model, or explicitly reset the parameters to their default values with

```
setparam("timefmt", "")
setparam("datefmt", "")
setparam("datetimefmt", "")
```

When using the *excel* driver for accessing Excel spreadsheets we need to be careful when reading times since these are passed as a real value that needs to be converted to Mosel's representation of times (the second half of the model working with strings remains unchanged).

```
declarations
 T: time
 D: date
 DT: datetime
 Dates: list of date
 r: real
end-declarations

! Select the format used by the spreadsheet
setparam("timefmt", "%0h:%0M:%0S %P")
setparam("datefmt", "%0m/%0d/%y")
setparam("datetimefmt", "%0d/%0m/%y %0H:%0M:%0S")

initializations from 'mmsheet.excel:datetime.xls'
 r as "skiph;Time1"               ! Time is stored as a real
 D as "skiph;Date1"
 DT as "skiph;DateTime1"
 Dates as "skiph;Dates"
end-initializations

T:=time(round(r*24*3600*1000))
writeln(D, ", ", T)
writeln(DT)
writeln(Dates)
```

For the *csv* driver only the second part of the model (reading from strings) is relevant since date and time values in CSV format files are always encoded as strings.

Our model implemented with SQL statements looks as follows.

```
model "Dates and times (SQL)"
 uses "mmsystem", "mmodbc"

 declarations
  T,: time
  D: date
  DT: datetime
  Dates: list of date
 end-declarations

 setparam("SQLverbose",true)
 SQLconnect("datetime.mdb")
 if not getparam("SQLsuccess"): setioerr("Database connection failed")

! Select the format used by the spreadsheet/database
! (database fields have date/time types)
 setparam("timefmt", "%y-%0m-%0d %0H:%0M:%0S")
 setparam("datefmt", "%y-%0m-%0d")
 setparam("datetimefmt", "%y-%0m-%0d %0H:%0M:%0S")

 T:=time(SQLreadstring("select * from Time1"))
 D:=date(SQLreadstring("select * from Date1"))
 DT:=datetime(SQLreadstring("select * from DateTime1"))
 SQLexecute("select * from Dates", Dates)

 setparam("timefmt", "%h:%0M %p")
 writeln(D, ", ", T)
 writeln(DT)
 writeln(Dates)

! Read date / time from strings (database fields have some string type)
 setparam("timefmt", "%Hh%0Mm")
 setparam("datefmt", "%dm%0my%0y")
 setparam("datetimefmt", "%dm%0my%0y, %Hh%0Mm")

 T:=time(SQLreadstring("select * from Time2"))
 D:=date(SQLreadstring("select * from Date2"))
 DT:=datetime(SQLreadstring("select * from DateTime2"))

 writeln(D, ", ", T)
 writeln(DT)

! Use Mosel's default format
 setparam("timefmt", "")
 setparam("datefmt", "")
 setparam("datetimefmt", "")

 writeln(D, ", ", T)
 writeln(DT)

 SQLexecute("delete from TimeOut")      ! Cleaning up previous results: works
 SQLexecute("delete from DateOut")      ! only for databases, cannot be used
 SQLexecute("delete from DateTimeOut")  ! with spreadsheets (instead, delete
                                        ! previous solutions directly in the
                                        ! spreadsheet file)
 SQLexecute("insert into TimeOut values (?)", [T])
 SQLexecute("insert into DateOut values (?)", [D])
 SQLexecute("insert into DateTimeOut values (?)", [DT])

 SQLdisconnect
end-model
```

6.12 Working with union types

A union is a container capable of holding an object of one of a predefined set of types. One possible use

of this functionality is for reading and storing input data of a-priori unknown type.

When initializing unions from text format data files only scalar values of basic types are considered. More precisely, integers, reals, and Booleans are assigned to the union; textual values are used to initialize the entity as if it was of the first compatible type of the union (for the union `any` this is a string). An I/O error will be raised if this type does not support initialization.

Similarly to input from text format files, when reading data from spreadsheets or databases into union entities only scalar values of basic types are considered. Moreover, the resulting type will depend on the type employed within the data source (*e.g.* many databases use a numeric type for storing Boolean values and some do not distinguish between integer and real values, all these types will therefore result in the Mosel type `real` when populating entities of type `any`; the *mmsheet.csv* driver and *diskdata* will read the Boolean constants true/false as textual types, whereas they are read with type `boolean` by the default text driver and the other spreadsheet drivers). Date and time types will be stored as `string` (exception: the *mmsheet.excel* driver returns a real for time data unless the cell is formatted as text in the spreadsheet) and need to be transformed applying the suitable format settings (see discussion in Section 6.11)

```
model "Union handling (ODBC)"
 uses 'mmodbc', 'mmsystem'

 declarations
   L,L2: list of any
   L3: list of text or real
   LS: list of text
 end-declarations

 setparam("SQLverbose",true)

 ! Reading data of different types from a database
 initializations from "mmodbc.odbc:debug;uniondata.sqlite"
   L as "UnionTab"
   L2 as "UnionLst"
   L3 as "UnionTab"
 end-initializations

 write("L orig: ")
 forall(i in L) write (i,": ", i.typeid, "; ")
 writeln
 ! Date and time types are read in textual form
 L(5).date:=date(text(L(5)))
 L(6).time:=time(text(L(6)))
 write("L new:  ")
 forall(i in L) write (i,": ", i.typeid, "; ")
 writeln

 ! Reading into a list defined with a restricted set of types
 write("L3:     ")
 forall(i in L3) write (i,": ", i.typeid, "; ")
 writeln

 ! Textual database types are always read as string
 write("L2:     ")
 forall(i in L2) write (i,": ", i.typeid, "; ")
 writeln

 LS:=sum(i in L) [text(i)]

 initializations to "mmodbc.odbc:debug;uniondata.sqlite"
  ! Writing data of type 'any' to a database table with various different types
   L as "UnionOut(IVal,RVal,BVal,SVal,DVal,TVal)"
  ! Writing data of type 'any' into textual fields of a database
   L as "Union2Out"
  ! Writing data of a union type to a database
```

```
   L3 as "UnionOut(IVal,RVal,BVal,SVal,DVal,TVal)"
  ! Writing text-format data to a database table with various different types
   LS as "UnionOut(IVal,RVal,BVal,SVal,DVal,TVal)"
 end-initializations

 writeln("Output to DB terminated.")

end-model
```

Assuming that the SQLite database tables have been created with these definitions:

```
SQLexecute("create table UnionTab (IVal integer, RVal real, BVal boolean, "+
  "SVal varchar(20), DVal date, TVal timestamp(3))")
SQLexecute("insert into UnionTab (IVal,RVal,SVal,BVal,TVal,DVal) values(?1,?2,?3,?4,?5,?6)",
  [ 5, 1.75, "some text", true, '11:25:30', '2021-03-20' ])
SQLexecute("create table UnionLst (UValues varchar(20))")
```

Then the program above displays the following output (notice the type change for the date and time in the second line):

```
L orig: 5: 1; 1.75: 2; 1: 1; some text: 3; 2021-03-20: 3; 11:25:30: 3;
L new:  5: 1; 1.75: 2; 1: 1; some text: 3; 2021-03-20: 13; 11:25:30: 14;
L3:     5: 2; 1.75: 2; 1: 2; some text: 11; 2021-03-20: 11; 11:25:30: 11;
L2:     5: 3; 1.75: 3; true: 3; some text: 3; 2021-03-20: 3; 11:25:30: 3;
```

With data input from text files in default Mosel format or using the *mmsheet.xsl/xslx* drivers the first line would display as

```
L orig: 5: 1; 1.75: 2; true: 4; some text: 3; 2021-03-20: 3; 11:25:30: 3;
```

and the definition of 'L3' needs to comprise the type `boolean` in order to be able to read the same data:

```
  L3: list of text or real or boolean
```

Note that when exporting unions, any non-scalar value or types that do not support conversion to string will result in a NIL value ('?' in text format files) in the generated file.

6.13 Working with dataframe formats

A *dataframe* is a 2-dimensional array that corresponds to a representation of data in table format with columns (fields) of different data types. The first array index is given by the row counter, the second index is derived from the field labels (the header row in a CSV or spreadsheet file) or in the absence of a header row in a CSV-format or spreadsheet file by a counter of the table columns.

6.13.1 Dataframe format for CSV

Dataframe-format CSV reading and writing is provided via the 'diskdata' functionality of the module *mmetc*, it is supported by the I/O driver *mmetc.diskdata* and equally by the subroutine `diskdata`. If the data types are not known upfront or may vary across the table columns the Mosel array type needs to be able to cover all possible cases, so typically this will be either `text` or a union type.

Assuming we wish to read this CSV file `mydata.csv` (note that this file contains some empty cells):

```
C_e,C_d,C_c,C_b,C_a,C_s
1,,"3",4,5.5,"r1"
6,7,"8",,10.5,"r2"
```

We can use the I/O driver *mmetc.diskdata* (the example also shows output into a newly created CSV file) to read data into fixed size or dynamic arrays of type `text` or of a suitable union type (here we simply use the predefined type `any`):

```
model 'dataframecsv'
 uses 'mmsystem', 'mmetc'

 declarations
   dfd: dynamic array(r:range, s:set of string) of text
   dff: array(rf:range, sf:set of string) of text
   dfa: dynamic array(ra:range, sa:set of string) of any
 end-declarations
 initialisations from 'mmetc.diskdata:'
   dfd as "dataframe;skiph;mydata.csv"
   dff as "dataframe;skiph;mydata.csv"
   dfa as "dataframe;skiph;mydata.csv"
 end-initialisations
 writeln("dyn:", dfd.size, " fix:", dff.size) ! Output displayed: dyn:10 fix:12

 initialisations to 'mmetc.diskdata:'
   dfd as "dataframe;skiph;res.csv"           ! Output CSV file with header line
   dfd as "dataframe;resnh.csv"               ! Output CSV file without header
 end-initialisations
end-model
```

Or we can employ the `diskdata` subroutine to perform the same tasks as shown in the following code snippet.

```
model 'dataframecsv'
 uses 'mmsystem', 'mmetc'

 declarations
   dfd: dynamic array(r:range, s:set of string) of text
   dff: array(rf:range, sf:set of string) of text
   dfa: dynamic array(ra:range, sa:set of string) of any
 end-declarations
 diskdata(ETC_DATAFRAME+ETC_SKIPH, "mydata.csv", dfd)
 diskdata(ETC_DATAFRAME+ETC_SKIPH, "mydata.csv", dff)
 diskdata(ETC_DATAFRAME+ETC_SKIPH, "mydata.csv", dfa)
 writeln("dyn:", dfd.size, " fix:", dff.size) ! Output displayed: dyn:10 fix:12

! Output CSV file with and without header line
 diskdata(ETC_SKIPH+ETC_OUT+ETC_CSV+ETC_DATAFRAME, "res.csv", dfd)
 diskdata(ETC_OUT+ETC_CSV+ETC_DATAFRAME, "resnh.csv", dfd)
end-model
```

If no header line is present in the CSV input file we need to adapt the type of the second index of the Mosel array storing the dataframe to be a `range`

```
 declarations
   dfd2: dynamic array(r21:range, r2:range) of text
 end-declarations
 initialisations from 'mmetc.diskdata:'
   dfd2 as "dataframe;datanh.csv"
 end-initialisations
! Same as:
! diskdata(ETC_DATAFRAME, "datanh.csv", dfd2)
 writeln("size df2=", dfd2.size)            ! Output displayed: size df2=10
```

With the I/O driver it is possible to select specific fields (this option is not available through the subroutine version):

```
declarations
  dfd3: dynamic array(r3:range,s3:set of string) of any
end-declarations
initialisations from 'mmetc.diskdata:'
  dfd3 as "dataframe;skiph;csv(C_d,C_b);mydata.csv"
end-initialisations
writeln("size df3=", dfd3.size)            ! Output displayed: size df3=2
```

6.13.2 Dataframe format for spreadsheets

The I/O drivers *xls*, *xlsx*, and *csv* of the module *mmsheet* implement reading and writing dataframe format data for spreadsheets and CSV files. Note that the CSV dataframe formats that are accessible through *mmsheet.csv* largely match the functionality of `diskdata` presented in the previous section, however, the linewise reading through `diskdata` usually is more efficient for large data files. The Mosel array type employed for representing the dataframe needs to be able to cover all possible cases, so typically this will be either `text` or a union type if the data types are not known upfront or may vary across the table columns.

Assuming we wish to read a spreadsheet file `mydata.xlsx` with the same contents the file `mydata.csv` in the previous section:

	A	B	C	D	E	F	G
1	**C_e**	**C_d**	**C_c**	**C_b**	**C_a**	**C_s**	
2	1		'3	4	5.5	r1	
3	6	7	'8		10.5	r2	
4							

We can use the `dataframe` option to read data into fixed size or dynamic arrays of type `text` or of a suitable union type (here we simply use the predefined type `any`). The example also shows output into newly created spreadsheet files:

```
model 'dataframesht'
 uses 'mmsystem', 'mmsheet'

 declarations
   dfd: dynamic array(r:range, s:set of string) of text
   dff: array(rf:range, sf:set of string) of text
   dfa: dynamic array(range, string) of any
 end-declarations
 initialisations from 'mmsheet.xlsx:mydata.xlsx'  ! Similarly for .xls or .csv
   dfd as "dataframe;skiph;[]"
   dff as "dataframe;skiph;[]"
   dfa as "dataframe;skiph;[]"
 end-initialisations
 writeln("dyn:", dfd.size, " fix:", dff.size) ! Output displayed: dyn:10 fix:12

 ! Typed data in 'dfa'
 writeln("entry (2,'C_d'): ", dfa(2,'C_d'), " has type ",
     dfa(2,'C_d').typeid)=integer.id )        ! Output displayed:  true
 writeln("entry (1,'C_s'): ", dfa(1,'C_s'), " has type ",
     dfa(1,'C_s').typeid)=string.id )         ! Output displayed:  true

 ! Writing out field names along with the array (option skiph+)
 initialisations to "mmsheet.xlsx:dfout.xlsx" ! Similarly for .xls or .csv
   dfa as 'dataframe;skiph+;[A:F]'            ! Output CSV file with header line
 end-initialisations

 ! Pre-populated re-ordered field names
 ! Writing the array without field names (option skiph)
```

```
 L:=["C_s","C_e","C_d","C_c","C_b","C_a"]
 initialisations to "mmsheet.xlsx:dfout2.xlsx"   ! Similarly for .xls or .csv
   L as '[A:F]'
   dfa as 'dataframe;skiph;[]'
 end-initialisations
end-model
```

If no header line is present in the input file we need to adapt the type of the second index of the Mosel array storing the dataframe to be a `range`

```
 declarations
   dfd2: dynamic array(r21:range, r2:range) of any
 end-declarations
 initialisations from 'mmsheet.xlsx:datanh.xlsx'  ! Similarly for .xls or .csv
   dfd2 as "dataframe;[]"
 end-initialisations
 writeln("size df2=", dfd2.size)               ! Output displayed: size df2=10
 initialisations to 'mmsheet.xlsx:dfoutnh.xlsx'   ! Similarly for .xls or .csv
   dfd2 as 'dataframe;[A:F]'
 end-initialisations
```

It is also possible to select specific fields in the sheet or within a given range, either using the field names or order numbers within the range:

```
declarations
   dfd3,dfd4: dynamic array(range,string) of any
 end-declarations
 initialisations from 'mmsheet.xlsx:mydata.xlsx'  ! Similarly for .xls or .csv
   dfd3 as "dataframe;skiph;[](C_d,C_b)"    ! Selection via field names
   dfd4 as "dataframe;skiph;[B:E](#1,#4)"   ! Selection by order number
 end-initialisations
 writeln("size df3=", dfd3.size)            ! Output displayed: size df3=2
 writeln("size df4=", dfd4.size)            ! Output displayed: size df4=3
```

6.13.3 Dataframe format for databases

The ODBC interface in *mmodbc* provides the subroutine `SQLdataframe` for reading all or selected fields of a database table into a Mosel array without any prior knowledge of data types or table dimensions: the expected argument is a 2-dimensional array with its first index a `range` and the second index a `set of string`, typically it will be either of type `text` or of a suitable union type. A typed array (such as `integer`) is accepted, but in this case only fields with matching types are read.

The following code snippet shows how to read all or selected columns of a database table into a Mosel array. It is possible to work with database commands that combine multiple tables—if this results in multiple occurrences of field names unique field names will be generated by appending counters.

```
model 'dataframedb'
 options keepassert
 uses 'mmodbc','mmsystem'

 declarations
  dfa: array(ra:range,csa:set of string) of any
  dft: array(rt:range,cst:set of string) of text
  dfa2: array(ra2:range,csa2:set of string) of any
 end-declarations

 SQLconnect("dbtest.sqlite")
 assert(getparam("SQLsuccess"))

! Read the entire table 'shirts'
 SQLdataframe("select * from shirts", dfa)
 writeln("csa: ", csa)   ! Output: csa: { `id', `style', `price', `color', `owner'}
```

```
 writeln(sum(i in ra) dfa(i,"owner").integer)

! Select some columns and specify new name for one of them
 SQLdataframe("select id as prod,color as couleur,style from shirts", dft)
 writeln("cst: ", cst)   ! Output: cst: { `prod', `couleur', `style'}

! Can handle data from multiple tables, in the case of multiple occurrences
! of field names unique names are generated
 SQLdataframe("select * from shirts inner join shop on id=article", dfa2)
 writeln("csa2: ", csa2)
   ! Output:  csa2: { `id', `style', `price', `color', `owner', `article', `owner_7'}

 SQLdisconnect
end-model
```

7 Trouble shooting

- *The module mmodbc cannot be initialized:* check whether the ODBC driver manager is installed and can be found and accessed by the application (in particular on Unix platforms: if the value returned for the control parameter `SQLdm` is negative then no driver manager has been found).
- *Missing ODBC driver:* the ODBC driver is a separate piece of software (not included in *mmodbc*) that is sometimes provided directly with the data source (typically the case under Windows), but in general it needs to be installed and set up separately.
- Connection strings should not contain any blanks. (This remark applies, for instance, to MySQL).
- **Spreadsheets are not databases.**
  - Close the spreadsheet file while executing a Mosel model that writes to it.
  - Make sure that the file is not opened in 'Read only' mode (this may happen, for instance, if several users access the file at the same time).
  - Columns in a spreadsheet range are not typed. The Excel ODBC driver scans the data in the first 8 rows to deduce the type of the data in every column. The number of rows scanned by the driver may be changed with the option `MAXSCANROWS` in the ODBC driver connection string.
  - You have to have enough space below the header line to fit in all the values you are going to write. So it is best to have nothing below the range.
  - To delete data from a range in an Excel spreadsheet you cannot just delete the entries in the cells (otherwise further data will be added after a blank rectangle). You have to remove completely the data rows and the enlarged range with the sequence *Edit* ≫ *Delete* ≫ *Shift cells up*.
- *Insertion failed:* check whether a key field is defined and the database table holds already data. Check the structure of the data table (sufficient number of columns, column names, field types). Close the database/spreadsheet before running the Mosel model (especially under Windows).
- *Reserved words:* Prod, Time, Value, Params *etc.* Inadvertent use of database keywords (depending on the database) often leads to difficult-to-diagnose error messages. We therefore suggest using longer, problem-specific names or 'myProducts', 'myParameters', *etc.* that will not clash with any reserved words. It is also possible to use quotes to indicate that a word should not be interpreted by the database; please refer to the documentation of your database for further detail (quotation characters are database-dependent).
- *Truncated strings:* the parameter `SQLcolsize`/`OCIcolsize` specifies the maximum string length for database fields, anything exceeding this size is cut off. If you observe any truncations, increase the value of this parameter, and possibly also augment the value of `SQLbufsize`/`OCIbufsize` accordingly.

8 SQL commands

The following SQL commands are used in the examples:

- select: `select *`: Sections 4.1.1, 6.2, 6.3, 6.5, select with specification of fields: Sections 4.1.1, 6.1, 6.5, 6.6, select with conditions: Section 6.7.
- insert: Sections 4.1.2, 6.1, 6.4, 6.6.
- create table: Sections 6.4, 6.6.
- delete: Section 4.1.2.
- drop table: Section 6.6.
- update: Sections 6.1, 6.6.