## Contents

1 Introduction 1
   1.1 What is Mosel? 1
   1.2 General Organization 1
   1.3 Running Mosel 3
   1.4 References 7
   1.5 Structure of this Manual 7

2 The Mosel Language 9
   2.1 Introduction 9
      Comments 9
      Identifiers 10
      Reserved Words 10
      Separation of Instructions, Line Breaking 10
      Conventions in this Document 11
   2.2 Structure of the Source File 11
   2.3 The Compiler Directives 12
   2.4 The Parameters Block 13
   2.5 Source File Inclusion 13
   2.6 The Declaration Block 14
      Elementary Types 15
      Sets 16
      Arrays 16
      Constants 17
   2.7 Expressions 18
      Introduction 18
      Aggregate Operators 19
      Arithmetic Expressions 20
      String Expressions 21
      Set Expressions 21
      Boolean Expressions 22
      Linear Constraint Expressions 22
      Tuples 23
   2.8 Statements 24
      Simple Statements 24
      Initialization Block 26
      Selections 29
      Loops 31
   2.9 Procedures and Functions 33
Definition ................................................................. 33
Formal Parameters: Passing Convention .......................... 34
Local Declarations ...................................................... 35
Overloading ................................................................. 36
Forward Declaration .................................................... 36

2.10 The public qualifier .............................................. 37
2.11 Handling of Input/Output ......................................... 37

3 Predefined Functions and Procedures ............................ 39
abs ................................................................. 40
arctan ............................................................. 41
bittest ............................................................... 42
ceil ............................................................... 43
cos ............................................................... 44
create .............................................................. 45
exists ............................................................. 46
exit ............................................................... 47
exp ............................................................... 48
exportprob ......................................................... 49
fclose ............................................................ 50
fflush ............................................................. 51
finalize ............................................................ 52
floor ............................................................... 53
fopen ............................................................. 54
fselect ............................................................ 55
fskipline ........................................................... 56
getact ............................................................. 57
getcoeff ........................................................... 58
getdual ............................................................. 59
getfid ............................................................. 60
getfirst ............................................................ 61
getlast ............................................................ 62
getobjval ........................................................... 63
getparam .......................................................... 64
getrcost ........................................................... 66
getsize ............................................................. 67
getslack ............................................................ 68
getsol .............................................................. 69
gettype ............................................................ 70
getvars ............................................................. 71
iseof ............................................................... 72
ishidden ............................................................ 73
makedir ................................................................. 147
removedir ........................................................... 148
system ................................................................. 149

9 mmxprs ................................................................. 151
  9.1 Control Parameters ........................................ 151
    XPRS_colorder .................................................. 151
    XPRS_loadnames .............................................. 152
    XPRS_problem .................................................. 152
    XPRS_probname .............................................. 152
    XPRS_verbose .................................................. 152
  9.2 Procedures and Functions ................................ 153
    clearmipdir .................................................... 154
    clearmodcut ................................................... 155
    delbasis ........................................................ 156
    getiis .......................................................... 157
    getlb ............................................................ 158
    getprobstat .................................................... 159
    getub ............................................................. 160
    initglobal ....................................................... 161
    loadbasis ....................................................... 162
    loadprob ......................................................... 163
    maximize, minimize ........................................... 164
    readbasis ......................................................... 166
    readdirs ........................................................ 167
    savebasis ......................................................... 168
    setcallback ...................................................... 169
    setlb ............................................................. 171
    setmipdir ......................................................... 172
    setmodcut ....................................................... 173
    setub ............................................................. 174
    writebasis ....................................................... 175
    writedirs ......................................................... 176
    writeprob ........................................................ 177
  9.3 Cut Pool Manager Routines ................................ 178
    addcut .......................................................... 179
    addcuts ........................................................... 180
    delcuts ........................................................... 181
    dropcuts .......................................................... 182
    getcnlist ........................................................ 183
    getcplist ........................................................ 184
    loadcuts .......................................................... 185
storecut. ......................................................... 186
storecuts ....................................................... 187

A Syntax Diagrams for the Mosel Language 189
  A.1 Main Structures and Statements. ......................... 189
  A.2 Expressions .............................................. 192

B Error Messages 195
  B.1 General Errors ............................................ 195
  B.2 Parser/compiler errors .................................... 198
      Errors Related to Modules ............................... 206
  B.3 Runtime Errors ........................................... 207
      Initializations ............................................ 207
      General Runtime Errors ................................. 207
      BIM Reader ............................................... 210
      Module Manager Errors ................................. 210

Index 213
How to Contact Dash

If you have any questions or comments on the use of Xpress-MP, please contact Dash technical support at:

**USA, Canada and The Americas**
Dash Optimization Inc.
560 Sylvan Avenue
Englewood Cliffs
NJ 07632
USA
*Telephone:* (201) 567 9445
*Fax:* (201) 567 9443
*email:* support-usa@dashoptimization.com

**Elsewhere**
Dash Optimization Ltd.
Quinton Lodge, Binswood Avenue
Leamington Spa
Warwickshire CV32 5RX
UK
*Telephone:* +44 1926 315862
*Fax:* +44 1926 315854
*email:* support@dashoptimization.com

If you have any sales questions or wish to order Xpress-MP software, please contact your local sales office, or Dash sales at:

**USA, Canada and The Americas**
Dash Optimization Inc.
560 Sylvan Avenue
Englewood Cliffs
NJ 07632
USA
*Telephone:* (201) 567 9445
*Fax:* (201) 567 9443
*email:* sales@dashoptimization.com

**Elsewhere**
Dash Optimization Ltd.
Blisworth House, Church Lane
Blisworth
Northants NN7 3BX
UK
*Telephone:* +44 1604 858993
*Fax:* +44 1604 858147
*email:* sales@dashoptimization.com

For the latest news and Xpress-MP software and documentation updates, please visit the Xpress-MP website at http://www.dashoptimization.com/

Last update June, 2003
1 Introduction

1.1 What is Mosel?

Mosel is an environment for modeling and solving problems. To this aim, it provides a language that is both a modeling and a programming language. The originality of the Mosel language is that there is no separation between a modeling statement (e.g. declaring a decision variable or expressing a constraint) and a procedure that actually solves the problem (e.g. call to an optimizing command). Thanks to this synergy, one can program a complex solution algorithm by combining modeling and solving statements.

Each category of problem comes with its own particular types of variables and constraints and a single kind of solver cannot be efficient in all cases. To take this into account, the Mosel system does not integrate any solver by default but offers a dynamic interface to external solvers provided as modules. Each solver module comes with its own set of procedures and functions that directly extends the vocabulary and capabilities of the Mosel language. The link between Mosel and a solving module is achieved at the memory level and does not require any modification of the core system.

This open architecture can also be used as a means to connect Mosel to other software. For instance, a module could define the functionality required to communicate with a specific database.

The modeling and solving tasks are usually not the only operations performed by a software application. This is why the Mosel environment is provided either in the form of libraries or as a standalone program.

1.2 General Organization

As input, Mosel expects a text file containing the source of the model/program to execute (henceforth we use just the term ‘model’ for ‘model/program’ except where there might be an ambiguity). This source file is first compiled by the Mosel compiler. During this operation, the syntax of the model is checked but no operation is executed. The result of the compilation is a Binary Model (BIM) that is saved in a second file. In this form, the model is ready to be executed and the source file is not required any more. To actually ‘run’ the model, the BIM file must be read in again by
Mosel and then executed. These different phases are handled by different modules that comprise the Mosel environment:

**The runtime library:** This library contains the Virtual Machine (VIMA) interpreter. It knows how to load a model in its binary format and how to execute it. It also implements a model manager (for handling several models at a time) and a Dynamic Shared Objects manager (for loading and unloading modules required by a given model). All the features of this library can be accessed from a user application.

**The compiler library:** The role of this module is to translate a source file into a binary format suitable for being executed by the VIMA Interpreter.

**The standalone application:** The ‘mosel’ application, also known as ‘Mosel Console’, is a command line interpreter linked to the two previous modules. It provides a single program to compile and execute models.

**Various modules:** These modules complete the Mosel set of functionalities by providing, for instance, optimization procedures. As an example, the ‘mmxprs’ module extends the Mosel language with the procedure ‘maximize’ that optimizes the current problem using the Xpress-Optimizer.

This modularized structure offers various advantages:

- Once compiled, a model can be run several times, for instance with different data sets, without the need for recompiling it.
- The compiled form of the program is system and architecture independent: it can be run on any operating system equipped with the Mosel runtime library and any modules required.
- The BIM file can be generated in order to contain no symbols at all. It is then safe, in terms of intellectual property, to distribute a model in its binary form.
- As a library, Mosel can be easily integrated into a larger application. The model may be provided as a BIM file and the application only linked to the runtime library.
- The Mosel system does not integrate any kind of solver but is designed in a way that a module can provide solving facilities. The direct consequence of this is that Mosel can be linked to different solvers and communicate with them directly through memory.
- This open architecture of Mosel makes extensions of the functionality possible on a case by case basis, without the need to modify the Mosel internals.
1.3 Running Mosel

The Mosel environment may be accessed either through its libraries or by means of two applications, perhaps the simplest of which is the Xpress-MP integrated visual environment, Xpress-IVE. Using a popular graphical interface, models can be developed and solved, providing simple access to all aspects of Mosel’s post-processing capabilities. Xpress-IVE is available under the Windows operating system only.

In its standalone version, Mosel offers a simple interface to execute certain generic commands either in batch mode or by means of a command line interpreter. The user may compile source models or programs (‘.mos’ files), load binary models (‘.bim’ files), execute them, display or save a matrix as well as the value of a given symbol. Several binary models can be loaded at a time and used alternatively.

The mosel executable accepts the following command line options:

- `-h` Display a short help message and terminate.
- `-V` Display the version number and terminate.
- `-s` Silent mode (valid only when running in batch mode)
- `-c commands` Run Mosel in batch mode. The parameter ‘commands’ must be a list of commands (see below) separated by semicolons (this list may have to be quoted with single or double quote depending on the operating system and shell being used). The ‘commands’ are executed in sequence until the end of the list or until an error occurs, then Mosel terminates. For example,

```
mose1 -c "cload -sg mymodel; run"
```

If no command line option is specified, Mosel starts in interactive mode. At the command prompt, the following commands may be executed (the arguments enclosed in square brackets [] are optional). The command line interpreter is case-insensitive, although we display commands in upper case for clarity:

**INFO** [symbol]: Without a parameter, this command displays information about the program being executed (this may be useful for problem reporting). Any parameter is interpreted as a symbol from the current model. If the requested symbol actually exists, this command displays some information about its type and structure.

**SYSTEM** command: Execute an operating system command.

Examples:
Execute the command ‘ls’ to display the current directory content and launch the VI editor to edit the file ‘mywork.mos’. Note that if the command contains blanks (usually the case if it requires parameters), quotes have to be used.

QUIT: Terminate the current Mosel session.

COMPILE [-sgep] filename [comment]: Compile the model ‘filename’ and generate the corresponding Binary Model (BIM) file if the compilation succeeds. The extension ‘.mos’ is appended to ‘filename’ if no extension is provided and the extension ‘.bim’ is used to form the name of the binary file. The flag ‘-e’ disables the automatic extension of the source file name. If the flag ‘-s’ is selected, the private object names (e.g. variables, constraints) are not saved into the BIM file. The flag ‘-g’ adds debugging information: it is required to locate a runtime error. The optional ‘comment’ parameter may be used to add a commentary to the BIM file (c.f. command ‘LIST’).

If the flag ‘-p’ is selected, only the syntax of the source file is checked, the compilation is not performed and no output file is generated.

Examples:

>compile mywork "This is an example"
>compile thismodel.mos

Compile the files ‘mywork.mos’ and ‘thismodel.mos’, creating the BIM files ‘mywork.bim’ and ‘thismodel.bim’ after successful completion of the compilation.

LOAD filename: Load the BIM file ‘filename’ into memory and open all modules it requires for later execution. The extension ‘.bim’ is appended to ‘filename’ if no extension is provided. If a model bearing the same name is already loaded in core memory it is replaced by the new one (the name of the model is specified by the statement model in the source file – it is not necessarily the file name).

Example:

>load mywork

Load ‘mywork.bim’ into memory (provided the source file begins with the statement ‘model mymodel’, the name of this problem is ‘mymodel’).

CLOAD [-sge] filename [comment]: Compile ‘filename’ then load the resulting file (if the compilation has succeeded). This command is equivalent to the consecutive execution of ‘compile filename’ and ‘load filename’. For an explanation of the options see command COMPILE.
**LIST:** Display the list of all models loaded using either CLOAD or LOAD. The information displayed for each model is:
- name: the model name (given by the model statement in the source file);
- number: the model number is automatically assigned when the model is loaded;
- size: the amount of memory used by the model (in bytes);
- system comment: a text string generated by the compiler indicating the source filename and if the model contains debugging information and/or symbols;
- user comment: the comment defined by the user at compile time (c.f. COMPILE, CLOAD).

The active model is marked by an asterisk (‘*’) in front of its name (the commands DELETE, RUN and RESET are applied to the active model). By default the last model that has been loaded is active.

**SELECT [number | name]:** Activate a model. The model can be selected using either its name or its order number. If no model reference is provided, information about the current active model is displayed.

**DELETE [number | name]:** Delete a model from memory (the BIM file is not affected by this command). If no model name or sequence number is given, the active model is deleted. If the active model is removed, the model loaded most recently (if any) becomes the new active model.

**RUN [parameters]:** Execute the active model. Optionally, a list of parameter values may be provided in order to initialize the parameters of the model and/or the control parameters of the modules used. The syntax of such an initialization is ‘param_name = value’ for a model parameter and ‘dsoname.ctrpar_name = value’, where dsoname is the name of a module and ctrpar_name the control parameter to set.

Examples:
> run ‘A=33,B=”word”,C=true,D=5.3e-5’
> run ‘Z=”aa”,mmxprs.XPRS_verbose=true’
> run T=1

**EXEC [-sge] filename [params]:** Compile ‘filename’, load, and then run the model. This command is equivalent to the consecutive execution of ‘cload filename’ and ‘run params’ except that the BIM file is not preserved. For an explanation of the options see command COMPILE.

**RESET:** Reinitialize the active model by releasing all the resources it uses.

**EXPORTPROB [-pms] [filename [objective]]:** Display or save to the given file (option ‘filename’) the matrix corresponding to the active problem. The matrix output
uses the LP format or the MPS format (flag ‘-m’). A problem is available after the execution of a model. The flags may be used to select the direction of the optimization (‘-p’: maximize), the file format (‘-m’: MPS format) and whether real object names should be used (‘-s’: scrambled names – this is the default if the object names are not available). The objective may also be selected by specifying a constraint name. When exporting matrices in MPS format any possibly specified lower bounds on semi-continuous or semi-continuous integer variables are lost. LP format matrices maintain the complete information.

DISPLAY symbol: Display the value of the given symbol. Before running the model, only constants can be accessed. For decision variables, the solution value is displayed (default 0); for constraints, it is the activity value (default 0).

SYMBOLS [-cspo]: Display the list of symbols published by the current model. The optional flags may be used to filter what kind of symbol to display: ‘-c’ for constants, ‘-s’ for subroutines, ‘-p’ for parameters and ‘-o’ for everything else.

LSLIBS: Display the list of all loaded dynamic shared objects (DSO) together with, for each module, its version number and its number of references (i.e. number of loaded models using it).

EXAMINE [-cspt] libname: Display the list of constants, procedures/functions, types and control parameters of the module libname. Optional flags may be used to select which information is displayed: ‘-c’ for constants, ‘-s’ for subroutines, ‘-t’ for types and ‘-p’ for control parameters.

FLUSHLIBS: Unload all unused dynamic shared objects.

If a command is not recognized, a list of possible keywords is displayed together with a short explanation. The command names can be shortened as long as there is no ambiguity (e.g. ‘cl’ can be used in place of CLOAD but ‘c’ is not sufficient because it could equally denote the COMPIL command). String arguments 1 may be quoted with either single or double quotes. Quoting is required if the text string starts with a digit or contains spaces and/or quotes.

Typically, a model will be loaded and executed with the following commands:

```
>cl mymodel
>run
```

If the BIM file is not required, the EXEC command may be preferred:

```
>exec model
```

1. The parameter 10 is a number, but “10” or ‘10’ are text strings.
1.4 References

Mosel could be described as an original combination of a couple of well known technologies. Here is a non-exhaustive list of the most important ‘originators’ of Mosel:

- The overall architecture of the system (compiler, virtual machine, native interface) is directly inspired by the Java language. Similar implementations are also commonly used in the languages for artificial intelligence (e.g. Prolog, Lisp).
- The syntax and the major building blocks of the Mosel language are in some aspects a simplification and for other aspects extensions of the Pascal language.
- The aggregate operators (like ‘sum’) are inherited from the ‘tradition of model builders’ and can be found in most of today’s modeling languages.
- The dynamic arrays and their particular link with sets are probably unique to Mosel but are at their origin a generalization of the sparse tables of the mp-model model builder.

1.5 Structure of this Manual

The main body of this manual is essentially organized into two parts. In Chapter 2, “The Mosel Language”, the basic building blocks of Mosel’s modeling and programming language are discussed.

Chapter 3, “Predefined Functions and Procedures” begins the reference section of this manual, providing a full description of all the functions and procedures defined as part of the core Mosel language. The functionality of the Mosel language may be expanded by loading modules: the following chapters describe the modules currently provided with the standard Mosel distribution: mmetc, mmive, mmodbc, mmquad, mmsystem and mmxprs.
2 The Mosel Language

The Mosel language can be thought of as both a modeling language and a programming language. Like other modeling languages it offers the required facilities to declare and manipulate problems, decision variables, constraints and various data types and structures like sets and arrays. On the other hand, it also provides a complete set of functionalities proper to programming languages: it is compiled and optimized, all usual control flow constructs are supported (selection, loops) and can be extended by means of modules. Among these extensions, optimizers can be loaded just like any other type of modules and the functionality they offer may be used in the same way as any Mosel procedures or functions. These properties make of Mosel a powerful modeling, programming and solving language with which it is possible to write complex solution algorithms.

The syntax has been designed to be easy to learn and maintain. As a consequence, the set of reserved words and syntax constructs has deliberately been kept small avoiding shortcuts and ‘tricks’ often provided by modeling languages. These facilities are sometimes useful to reduce the size of a model source (not its readability) but also are likely to introduce inconsistencies and ambiguities in the language itself, making it harder to understand and maintain. The major benefit of this rigour is that when a rule is established, it is valid everywhere in the language. For instance, wherever a set is expected, any kind of set expression is accepted.

2.1 Introduction

Comments

A comment is a part of the source file that is ignored by the compiler. It is usually used to explain what the program is supposed to do. Either single line comments or multi lines comments can be used in a source file. For the first case, the comment starts with the ‘!’ character and terminates with the end of the line. A multi-line commentary must be inclosed in ‘(!’ and ‘!)’. Note that it is possible to nest several multi-line commentaries.

! In a comment
This text will be analysed
(! Start of a multi line
    (! another comment
    blabla
Comments may appear anywhere in the source file.

**Identifiers**

Identifiers are used to name objects (variables, for instance). An identifier is an alphanumeric (plus \_\_) character string starting with an alphabetic character or \_\. All characters of an identifier are significant and the case is important (the identifier ‘word’ is not equivalent to ‘Word’).

**Reserved Words**

The reserved words are identifiers with a particular meaning that determine a specific behaviour within the language. Because of their special role, these *keywords* cannot be used to name user defined objects (i.e. they cannot be redefined). The list of reserved words is:

```
and, array, as, boolean, break, case, declarations, div, do, mpvar,
dynamic, elif, else, end, false, forall, forward, from, function, if, in,
include, initialisations, initializations, integer, inter,
is_binary, is_continuous, is_free, is_integer, is_partint,
is_semcont, is_semint, is_sos1, is_sos2, linctr, max, min, mod, model,
next, not, of, options, or, parameters, procedure, public, prod, range,
real, repeat, set, string, sum, then, to, true, union, until, uses, while.
```

Note that, although the lexical analyser of Mosel is case-sensitive, the reserved words are defined both as lower and upper case (i.e. ‘AND’ and ‘and’ are keywords but not ‘And’).

**Separation of Instructions, Line Breaking**

In order to improve the readability of the source code, each statement may be split across several lines and indented using as many spaces or tabulations as required. However, as the line breaking is the expression terminator, if an expression is to be split, it must be cut after a symbol that implies a continuation like an operator (+, -, \-, \..., \_) or a comma (,,) in order to warn the analyser that the expression continues in the following line(s).
Moreover, the character ‘;’ can be used as an expression terminator.

```mosel
A+B ; -C+D // 2 expressions on the same line
```

Some users prefer to explicitly mark the end of each expression with a particular symbol. This is possible using the option explterm (see Section 2.3, “The Compiler Directives”) which disables the default behaviour of the compiler. In that case, the line breaking is not considered any more as an expression separator and each statement finishing with an expression must be terminated by the symbol ‘;’.

```mosel
A+B;    // expression 1
-C+D;   // expression 2
A+B     // expression 3...
-C+D;   // ...end of expression 3
```

**Conventions in this Document**

In the following sections, the language syntax is explained. In all code templates, the following conventions are employed:

- **word**: ‘word’ is a keyword and should be typed as is;
- **todo**: ‘todo’ is to be replaced by something else that is explained later;
- **[ something ]**: ‘something’ is optional and the entire block of instructions may be omitted;
- **[ something ...]**: ‘something’ is optional but if used, it can be repeated several times.

### 2.2 Structure of the Source File

The general structure of a Mosel source file is as follows:

```mosel
model model_name
[ Directives ]
[ Parameters ]
[ Body ]
end-model
```

```
The ‘model’ statement marks the beginning the program and the statement ‘end-model’ its end. Any text following this instruction is ignored (this can be used for adding plain text comments after the end of the program). The model name may be any quoted string or identifier, this name will be used as the model name in the Mosel model manager. An optional set of ‘directives’ and a ‘parameters’ block may follow. The actual program/model is described in the ‘body’ of the source file which consists of a succession of declaration blocks, subroutine definitions and statements. It is important to understand that the language is ‘procedural’ and not ‘declarative’: the declarations and statements are compiled and executed in the order of their appearance. As a consequence, it is not possible to refer to an identifier that is declared later in the source file or consider that a statement located later in the source file has already been executed. Moreover, the language is ‘compiled’ and not ‘interpreted’: the entire source file is first translated – as a whole – into a binary form (the ‘BIM file’), then this binary form of the program is read again to be executed. During the compilation, except for some simple constant expressions, no action is actually performed. This is why only some errors can be detected during the compilation time, any others being detected when running the program.

2.3 The Compiler Directives

The compiler accepts two different types of directives: the ‘uses’ statement and the ‘options’ statement.

The general form of a ‘uses’ statement is:

`uses libname1 [, libname2 ...] [;]`

This clause asks the compiler to load the listed modules and import the symbols they define.

The compiler options may be used to modify the default behaviour of the compiler. The general form of an ‘options’ statement is:

`options optname1 [, optname2 ...]`

The supported options are:

- `explterm`: asks the compiler to expect explicit expression termination (see the section, “Separation of Instructions, Line Breaking”)
- `noimplicit`: enables the implicit declarations (see the section “About Implicit Declarations”)
For example,

```mos
uses 'mmsystem'
options noimplicit, explterm
```

### 2.4 The Parameters Block

A model parameter is a symbol, the value of which can be set just before running the model (optional parameter of the ‘run’ command of the command line interpreter). The general form of the parameters block is:

```
parameters
  ident1 = Expression1
  [ ident2 = Expression2 ...]
end-parameters
```

where each identifier `identi` is the name of a parameter and the corresponding expression `Expressioni` its default value. This value is assigned to the parameter if no explicit value is provided at the start of the execution of the program (e.g. as a parameter of the ‘run’ command). Note that the type (integer, real, text string or Boolean) of a parameter is implied by its default value. Model parameters are manipulated as constants in the rest of the source file (it is not possible to alter their original value).

```mos
parameters
  size=12    ! integer parameter
  R=12.67    ! real parameter
  F="myfile" ! text string parameter
  B=true     ! Boolean parametersize=12
end-parameters
```

### 2.5 Source File Inclusion

A Mosel program may be split into several source files by means of file inclusion. The ‘include’ instruction performs this task:

```
include filename
```

where `filename` is the name of the file to be included. If this file name has no extension, the extension ‘.mos’ is automatically appended to the string.
The ‘include’ instruction is replaced at compile time by the contents of the file filename.

Assuming the file ‘a.mos’ contains:

```
model "Example for file inclusion"
  writeln(‘From the main file’)
  include "b"
end-model
```

And the file ‘b.mos’:

```
writeln(‘From an included file’)
```

Due to the inclusion of ‘b.mos’, the file ‘a.mos’ is equivalent to:

```
model "Example for file inclusion"
  writeln(‘From the main file’)
  writeln(‘From an included file’)
end-model
```

Note that file inclusion cannot be used inside blocks of instructions or before the body of the program (as a consequence, a file included cannot contain any of the following statements: uses, options or parameters).

## 2.6 The Declaration Block

The role of the declaration block is to give a name, a type, and a structure to the entities that the processing part of the program/model will use. The type of a value defines its domain (for instance integer or real) and its structure, how it is organised, stored (for instance a reference to a single value or an ordered collection in the form of an array). The declaration block is composed of a list of declaration statements enclosed between the instructions declarations and end-declarations.

```
declarations
  Declare_stat
  [ Declare_stat ...]
end-declarations
```

Several declaration blocks may appear in a single source file but a symbol introduced in a given block cannot be used before that block. Once a name has been assigned to an entity, it cannot be reused for anything else.
Elementary Types

Elementary objects are used to build up more complex data structures like sets or arrays. It is, of course, possible to declare an entity as a reference to a value of one of these elementary types. Such a declaration looks as follows:

\[ \text{ident1 [, ident2 ...]} : \text{type}\_\text{name} \]

where \text{type}\_\text{name} is the type of the objects to create. Each of the identifiers \text{identi} is then declared as a reference to a value of the given type.

The type name may be either a basic type (integer, real, string, boolean), an MP type (mpvar, linctr) or an external type. MP types are related to Mathematical Programming and allow declaration of decision variables and linear constraints. Note that the linear constraint objects can also be used to store linear expressions. External types are defined by modules (the documentation of each module describes how to use the type(s) it implements).

```
declarations
i,j:integer
str:string
x,y,z:mpvar
end-declarations
```

Basic Types

The basic types are:

- integer: an integer value between \(-214783648\) and \(2147483647\)
- real: a real value between \(-1.7e+308\) and \(1.7e+308\).
- string: some text.
- boolean: the result of a Boolean (logical) expression. The value of a Boolean entity is either the symbol true or the symbol false.

After its declaration, each entity receives an initial value of 0, an empty string, or false depending on its type.

MP Types

Two special types are provided for mathematical programming.

- mpvar: a decision variable
- linctr: a linear constraint
Sets

Sets are used to group a collection of elements of a given type. Declaring a set consists of defining the type of elements to be collected.

The general form of a set declaration is:

\[ \text{ident1} [\text{\ldots}], \text{ident2} \ldots]: \text{set of type\_name} \]

where \text{type\_name} is one of the elementary types. Each of the identifiers \text{identi} is then declared as a set of the given type.

A particular set type is also available that should be preferred to the general form wherever possible because of its better efficiency: the range set is a collection of consecutive integers in a given interval. The declaration of a range set is achieved by:

\[ \text{ident1} [\text{\ldots}], \text{ident2} \ldots]: \text{range [set of integer]} \]

Each of the identifiers \text{identi} is then declared as a range set of integers.

Every newly created set is empty.

```mosel
declarations
  s1: set of string
  r1: range
end-declarations
```

Arrays

An array is a collection of labelled objects of a given type. A label is defined by a list of indices taking their values in domains characterised by sets: the indexing sets. An array may be either of fixed size or dynamic. For fixed size arrays, the size (i.e. the total number of objects it contains, or cells) is known when it is declared. All the required cells (one for each object) are created and initialized immediately. Dynamic arrays are created empty. The cells are created when they are assigned a value (c.f. the section ”Assignment”) and the array may then grow ‘on demand’. The value of a cell that has not been created is the default initial value of the type of the array. The general form of an array declaration is:

\[ \text{ident1} [\text{\ldots}], \text{ident2} \ldots]: \text{[dynamic] array(list\_of\_sets) of type\_name} \]

where \text{list\_of\_sets} is a list of set declarations/expressions separated by commas and \text{type\_name} is one of the elementary types. Each of the identifiers \text{identi} is then declared as an array of the given type and indexed by the given sets. In the list of indexing sets, a set declaration can be anonymous (i.e. \text{rs\_set \ of real} can be
replaced by set of real if no reference to rs is required) or shortened to the type of the set (i.e. set of real can be replaced by real in that context).

```mosel
declarations
  e: set of string
  t1: array ( e, rs: set of real, range, integer ) of real
  t2: array ( ("i1","i2"), 1..3 ) of integer
end-declarations
```

By default, an array is of fixed size if all of its indexing sets are of fixed size (i.e. they are either constant or finalized (c.f. procedure finalize)). Otherwise, it is dynamic. The qualifier dynamic may be used to force an array to be dynamic.

Note that once a set is employed as an indexing set, Mosel makes sure that its size is never reduced in order to guarantee that no entry of any array becomes inaccessible. Such a set is called ‘fixed’.

**Special case of dynamic arrays of type mpvar**

If an array of type mpvar is defined as dynamic or the size of at least one of its indexing sets is unknown at declaration time (i.e. empty set), the corresponding variables are not created. In that case, it is required to create each of the relevant entries of the array by using the create procedure as there is no way to assign a value to a decision variable.

**Constants**

A constant is an identifier for which the value is known at declaration time and that will never be modified. The general form of a constant declaration is:

```mosel
identifier = Expression
```

where `identifier` is the name of the constant and `Expression` its initial and only value. The expression must be of one of the basic types or a set of one of these types.

```mosel
declarations
  STR='my const string'
  I1=12
  R=1..10
  S={2.3,5.6,7.01}
end-declarations
```
2.7 Expressions

Expressions are, together with the keywords, the major building blocks of a language. This section summarises the different basic operators and connectors used to build expressions.

Introduction

Expressions are constructed using constants, operators and identifiers (of objects or functions).

If an identifier appears in an expression its value is the value referenced by this identifier. In the case of a set or an array, it is the whole structure. To access a single cell of an array, it is required to ‘dereference’ this array. The dereferencing of an array is denoted as follows:

```
array_ident (Exp1 [ , Exp2 ...])
```

where `array_ident` is the name of the array and `Expi` an expression of the type of the `i^{th}` indexing set of the array. The type of such an expression is the type of the array and its value the value stored in the array with the label ‘`Exp1 [ , Exp2 ...]`’.

A function call is denoted as follows:

```
function_ident
or
function_ident (Exp1 [ , Exp2 ...])
```

where `function_ident` is the name of the function and `Expi` the `i^{th}` parameter required by this function. The first form is for a function requiring no parameter.

The special function `if` allows one to make a selection among expressions. Its syntax is the following:

```
if (Bool_expr , Exp1 , Exp2)
```

which evaluates to `Exp1` if `Bool_expr` is `true` or `Exp2` otherwise. The type of this expression is the type of `Exp1` and `Exp2` which must be of the same type.

The Mosel compiler operates automatic conversions to the type required by a given operator in the following cases:

- in the dereference list of an array:
  ```
  integer → real;
  ```
• in a function or procedure parameter list:
  
  integer → real, linctr;
  real → linctr;
  mpvar → linctr;

• anywhere else:
  
  integer → real, string, linctr;
  real → string, linctr;
  mpvar → linctr;
  boolean → string.

It is possible to force a basic type conversion using the type name as a function (i.e. integer, real, string, boolean). In the case of string, the result is the textual representation of the converted expression. In the case of boolean, for numerical values, the result is true if the value is nonzero and for strings the result is true if the string is the word ‘true’. Note that explicit conversions are not defined for MP types, and structured types (e.g. ‘linctr(x)’ is a syntax error).

! Assuming A=3.5, B=2
  integer(A+B) ! = 5
  string(A-B) ! = "1.5"
  real(integer(A+B)) ! = 5.5 (because the compiler simplifies the expression)

Parentheses may be used to modify the predefined evaluation order of the operators or simply to group subexpressions.

Aggregate Operators

An operator is said to be ‘aggregate’ when it is associated to a list of indices for each of which a set of values is defined. This operator is then applied to its operands for each possible tuple of values (e.g. the summation operator sum is an aggregate operator).

The general form of an aggregate operator is:

\[ Aggregate\_ident \ (\text{Iterator1} \ [, \ \text{Iterator2} \ ...]) \ Expression \]

where the Aggregate\_ident is the name of the operator and Expression an expression compatible with this operator (see below for the different available operators). The type of the result of such an aggregate expression is the type of Expression.
An iterator is one of the following constructs:

\[
\begin{align*}
&\text{Set_expr} \\
\text{or} \\
&\text{ident}_1 [, \text{ident}_2 \ldots] \text{ in Set_expr} [ \mid \text{Bool_expr}] \\
\text{or} \\
&\text{ident} = \text{Expression} [ \mid \text{Bool_expr}] \\
\end{align*}
\]

The first form gives the list of the values to be taken without specifying an index name. With the second form, the indices named \text{ident}_i take successively all values of the set defined by \text{Set_expr}. With the third form, the index \text{ident} is assigned a single value (which must be a scalar). For the last two cases, the scope of the created identifier is limited to the scope of the operator (i.e. it exists only for the following iterators and for the operand of the aggregate operator). Moreover, an optional condition can be stated by means of \text{Bool_expr} which can be used as a filter to select the relevant elements of the domain of the index. It is important to note that this condition is evaluated as early as possible. As a consequence, a Boolean expression that does not depend on any of the defined indices in the considered iterator list is evaluated only once, namely \textit{before} the aggregate operator itself and not for each possible tuple of indices.

An index is considered to be a constant: it is not possible to change explicitly the value of a named index (using an assignment for instance).

**Arithmetic Expressions**

Numerical constants can be written using the common scientific notation. Arithmetic expressions are naturally expressed by means of the usual operators (+, -, *, / division, unary -, unary +, ^ raise to the power). For integer values, the operators \text{mod} (remainder of division) and \text{div} (integral division) are also defined. Note that \text{mpvar} objects are handled like real values in expression.

The \text{sum} (summation) aggregate operators is defined on integers, real and mpvar. The aggregate operators \text{prod} (product), \text{min} (minimum) and \text{max} (maximum) can be used on integer and real values.

\[
\begin{align*}
x\ast 5.5 + (2+z)^4 + \cos(12.4) \\
\text{sum}(i \text{ in } 1..10) \ (\text{min}(j \text{ in } s) \ t(i) \ast (a(j) \text{ mod } 2))
\end{align*}
\]
String Expressions

Constant strings of characters must be quoted with single (') or double quote ("). Strings enclosed in double quotes may contain C-like escape sequences introduced by the ‘backslash’ character (\a \b \f \n \r \t \v).

Each sequence is replaced by the corresponding control character (e.g. \n is the ‘new line’ command) or, if no control character exists, by the second character of the sequence itself (e.g. \ is replaced by '\').

The escape sequences are not interpreted if they are contained in strings that are enclosed in single quotes.

Example:

'c:\ddd1\ddd2\ddd3' is understood as c:\ddd1\ddd2\ddd3
"c:\ddd1\ddd2\ddd3" is understood as c:ddd1ddd2ddd3

There are two basic operators for strings: the concatenation, written ‘+’ and the difference, written ‘-’.

"a1b2c3d5"+"e6"    ! = "a1b2c3d5e6"
'a1b2c3d5'-'3d5"   ! = "a1b2c"

Set Expressions

Constant sets are described using one of the following constructs:

{ [ Exp1 [, Exp2 ...]]

or

[] Integer_exp1 .. Integer_exp2 []

The first form enumerates all the values contained in the set and the second form, restricted to sets of integers, gives an interval of integer values. This form implicitly defines a range set.

The basic operators on sets are the union written +, the difference written – and the intersection written *.

The aggregate operators union and inter can also be used to build up set expressions.

{(1,2,3)+(4,5,6)-(5..8)*(6,10)  ! = {1,2,3,4,5}
('a','b','c')*('b','c','d')    ! = {'b','c'}

"a1b2c3d5"+"e6"    ! = "a1b2c3d5e6"
'a1b2c3d5'-'3d5"   ! = "a1b2c"
Boolean Expressions

A Boolean expression is an expression whose result is either true or false. The traditional comparators are defined on integer and real values: <, <=, =, <> (not equal), >=, >.

These operators are also defined for string expressions. In that case, the order is defined by the ISO-8859-1 character set (i.e. roughly: punctuation < digits < capitals < small letters < accented letters).

With sets, the comparators <= ('is subset of'), >= ('is superset of'), = ('equality of contents') and <> ('difference of contents') are defined. These comparators must be used with two sets of the same type. Moreover, the operator 'expr in Set_expr' is true if the expression expr is contained in the set Set_expr. The opposite, the operator not in is also defined.

To combine boolean expressions, the operators and (logical and) and or (logical or) as well as the unary operator not (logical negation) can be used. The evaluation of an arithmetic expression stops as soon as its value is known.

The aggregate operators and and or are the natural extension of their binary counterparts.

3<=x and y>=45 or t<>r and not r in {1..10}
and(i in 1..10) 3=x(i)

Linear Constraint Expressions

Linear constraints are built up using linear expressions on the decision variables (type mpvar).

The different forms of constraints are:

\[
\text{Linear_expr} \\
\text{or} \\
\text{Linear_expr1 Ctr_cmp Linear_expr2} \\
\text{or} \\
\text{Linear_expr SOS_type} \\
\text{or} \\
\text{mpvar_ref mpvar_type1}
\]
or
\[ mpvar_ref\ mpvar_type2\ \text{Arith}\_\text{expr} \]

In the case of the first form, the constraint is 'unconstrained' and is just a linear expression. For the second form, the valid comparators are \(\leq, \geq, =\). The third form is used to declare special ordered sets. The types are then \(\text{is\_sos1}\) and \(\text{is\_sos2}\). The coefficients of the variables in the linear expression are used as weights for the SOS (as a consequence, a 0-weighted variable cannot be represented this way, procedure \text{makesos1} or \text{makesos2} has to be used instead). The last two types are used to set up special types for decision variables. The first series does not require any extra information: \(\text{is\_continuous}, \text{is\_integer}, \text{is\_binary}, \text{is\_free}\). The second series of types is associated with a threshold value stated by an arithmetic expression: \(\text{is\_partint}\) for partial integer, the value indicates the limit up to which the variable must be integer, above which it is continuous. For \(\text{is\_semcont}\) (semi-continuous) and \(\text{is\_semint}\) (semi-continuous integer) the value gives the semi-continuous limit of the variable (that is, the lower bound on the part of its domain that is continuous or consecutive integers respectively). Note that these constraints on single variables are also considered as common linear constraints.

\[
3*y+\text{sum}(i\ \text{in} \ 1..10)\ x(i)*i \geq z-t
\]

\(t\ \text{is\_integer}\) ! Define an integer variable
\(t \geq 7\) ! Lower bound on \(t\): \(t=7,8,\ldots\)
\(\text{sum}(i\ \text{in} \ 1..10)\ i*x(i)\ \text{is\_sos1}\) ! SOS1 \(\{x(1),x(2),\ldots\}\) with
\(\text{weights}\ 1,2,\ldots\)
\(y\ \text{is\_partint} 5\) ! \(y=0\) or \(y=5,6,\ldots\)
\(y \leq 20\) ! Upper bound on \(y\): \(y=0\) or \(y=5,6,\ldots,20\)

Internally all linear constraints are stored in the same form: a linear expression (including a constant term) and a constraint type (the right hand side is always 0). This means, the constraint expression \('3*x>=5*y-10'\) is internally represented by: \(3*x-5*y+10\) and the type 'greater than or equal to'. When a reference to a linear constraint appears in an expression, its value is the linear expression it contains. For example, if the identifier 'ctl' refers to the linear constraint \(3*x>=5*y-10\), the expression \('z-x+ctl'\) is equal to: \('z-2*x-5*y+10'\).

Note that the value of an unary constraint of the type \('x\ \text{is\_type\ threshold}'\) is \('x - \text{threshold}'\) and the value of \('x\ \text{is\_integer}'\) is \('x - \text{MAX\_INT}'\).

Tuples

A tuple is a list of expressions enclosed in square brackets.

\([\text{Exp1}[, \text{Exp2} \ldots]]\)
Tuples are essentially used to initialize arrays. They can also be employed as replacement for arrays in function or procedure parameters.

```mosel
declarations
T:array(1..2,1..3) of integer
end-declarations
T:=[1,2,3,4,5,6]
writeln(T)                ! displays: [1,2,3,
!            4,5,6]
writeln(getsize([2,3,4])) ! displays: 3
```

There is no operator defined on tuples and it is not possible to declare an identifier which takes a tuple as its value.

### 2.8 Statements

Four types of statements are supported by the Mosel language. The simple statements can be seen as elementary operations. The initialization block is used to load data from a file or save data to a file. Selection statements allow one to choose between different sets of statements depending on conditions. Finally, the loop statements are used to repeat operations.

Each of these constructs is considered as a single statement. A list of statements is a succession of statements. No particular statement separator is required between statements except if a statement terminates by an expression. In that case, the expression must be finished by either a line break or the symbol `;`.

#### Simple Statements

**Assignment**

An ‘assignment’ consists in changing the value associated to an identifier. The general form of an assignment is:

- `ident_ref := Expression`
- `ident_ref += Expression`
- `ident_ref -= Expression`
where \( \text{ident\_ref} \) is a reference to a value (i.e. an identifier or an array dereference) and \( \text{Expression} \) is an expression of a compatible type with \( \text{ident\_ref} \). The ‘direct assignment’, denoted \( := \), replaces the value associated with \( \text{ident\_ref} \) by the value of the expression. The ‘additive assignment’, denoted \( += \), and the ‘subtractive assignment’, denoted \( -= \), are basically combinations of a direct assignment with an addition or a subtraction. They require an expression of a type that supports these operators (for instance it is not possible to use additive assignment with Boolean objects).

The additive and subtractive assignments have a special meaning with linear constraints in the sense that they preserve the constraint type of the assigned identifier: normally a constraint used in an expression has the value of the linear expression it contains, the constraint type is ignored.

\[
\begin{align*}
c &:= 3x+y \geq 5 \\
c &:= c+y & \text{! implies } c &\geq 3x+2y-5 \geq 0 \\
c &:= 3x+y \geq 5 \\
c &:= c + y & \text{! implies } c &\geq 3x+2y-5 \text{ (c becomes unconstrained)}
\end{align*}
\]

The direct assignment may also be employed to initialize arrays using tuples.

\[
\begin{align*}
\text{declarations} \\
T &\text{:array(1..10) of integer} \\
\text{end-declarations} \\
T &:= [2,4,6,8] \quad \text{! } T(1):=2; T(2):=4;... \\
T(2) &:= [7,8,9,19] \quad \text{! } T(2):=7; T(3):=8;...
\end{align*}
\]

**About Implicit Declarations**

Each symbol should be declared before being used. However, an implicit declaration is issued when a new symbol is assigned a value the type of which is unambiguous.

\[
\begin{align*}
! \text{ Assuming } A, S, SE \text{ are unknown symbols} \\
A &:= 1 & \text{! } A \text{ is automatically defined} \\
&\quad \text{! } \text{ as an integer reference} \\
S &:= \{1,2,3\} & \text{! } S \text{ is automatically defined} \\
&\quad \text{! } \text{ as a set of integers} \\
SE &:= {} & \text{! This produces a parser error as} \\
&\quad \text{! } \text{ the type of SE is unknown}
\end{align*}
\]

In the case of arrays, the implicit declaration should be avoided or used with particular care as Mosel tries to deduce the indexing sets from the context and
decides automatically whether the created array must be dynamic. The result is not necessarily what is expected.

\[
\begin{align*}
A(1) &:= 1 \quad ! \text{implies: } A: \text{array}(1..1) \text{ of integer} \\
A(t) &:= 2.5 \quad ! \text{assuming } \{ t \in 1..10 | f(t) > 0 \} \\
&\quad ! \text{implies: } A: \text{dynamic array}(\text{range}) \text{ of real}
\end{align*}
\]

The option \texttt{noimplicit} disables implicit declarations.

\section*{Linear Constraint Expression}

A linear constraint expression can be assigned to an identifier but can also be stated on its own. In that case, the constraint is said to be ‘anonymous’ and is added to the set of already defined constraints. The difference from a ‘named constraint’ is that it is not possible to refer to an anonymous constraint again, for instance to modify it.

\[
10 \leq x; \ x \leq 20 \\
x \text{ is_integer}
\]

\section*{Procedure Call}

Not all required actions are coded in a given source file. The language comes with a set of predefined procedures that perform specific actions (like displaying a message). It is also possible to import procedures from external locations by using modules (c.f. “The Compiler Directives”).

The general form of a procedure call is:

\[
\begin{align*}
\text{procedure_ident} \\
\text{procedure_ident} \ (\text{Exp}_1 [, \ \text{Exp}_2 \ldots])
\end{align*}
\]

where \textit{procedure_ident} is the name of the procedure and, if required, \textit{Exp}_i is the \(i\)th parameter for the call. Refer to Chapter 3, “Predefined Functions and Procedures” of this manual for a comprehensive listing of the predefined procedures. The modules documentation should also be consulted for explanations about the procedures provided by each module.

\[
\text{writeln("hello!"}) \quad ! \text{displays the message: } \text{hello!}
\]

\section*{Initialization Block}

The initialization block may be used to initialize objects (scalars, arrays or sets) of basic type from text files or to save the values of such objects to text files. Scalars and
arrays of external types supporting this feature may also be initialized using this facility.

The first form of an initialization block is used to initialize data from a file:

```
initializations from Filename
  ident1 [as Label1]
  or
  [identT11, identT12 [ , IdentT13 ]] as LabelT1
  [ ident2 [as Label2]
  or
  [identT21, identT22 [ , IdentT23 ]] as LabelT2
  ]
end-initializations
```

where Filename, a string expression, is the name of the file to read, identi any object identifier and identTij an array identifier. Each identifier is automatically associated to a label: by default this label is the identifier itself but a different name may be specified explicitly using a string expression Labeli. When an initialization block is executed, the given file is opened and the requested labels are searched for in this file to initialize the corresponding objects. Several arrays may be initialized with a single record. In this case they must be all indexed by the same sets and the label is obligatory. After the execution of an ‘initializations from’ block, the control parameter ‘nbread’ reports the number of items actually read in.

An initialization file must contain one or several records of the following form:

```
Label: value
```

where Label is a text string and value either a constant of a basic type (integer, real, string or boolean) or a collection of values separated by spaces and enclosed in square brackets. Collections of values are used to initialize sets or arrays – if such a record is requested for a scalar, then the first value of the collection is selected. When used for arrays, indices enclosed in round brackets may be inserted in the list of values to specify a location in the corresponding array.

Note also that:

- no particular formatting is required: spaces, tabulations, and line breaks are just normal separators
- the special value ‘*’ implies a no-operation (i.e. the corresponding entity is not initialized)
• single line comments are supported (i.e. starting with ‘!’ and terminated by the end of the line)

• Boolean constants are either the identifiers ‘false’ and ‘true’ or the numerical constants ‘0’ and ‘1’

• all text strings (including the labels) may be quoted using either single or double quotes. In the latter case, escape sequences are interpreted (i.e. use of ‘\’).

The second form of an initialization block is used to save data to a file:

```mosel
initializations to Filename
      ident1 [as Label1]
    or
      [identT11, identT12 [, IdentT13 ...]] as LabelT1
      ident2 [as Label2]
    or
      [identT21, identT22 [, IdentT23 ...]] as LabelT2
...
end-initializations
```

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

For example, assuming the file ‘a.dat’ contains:

```mosel
! Example of the use of initialization blocks
t:[ (1 un) [10 11] (2 deux) [* 22] (3 trois) [30 33]]
t2:[ 10 (4) 30 40 ]
’nb used’ : 0
```

consider the following program:

```mosel
model "Example initblk"
declarations
  nb_used:integer
  s: set of string
```

1. A copy of the original file is saved prior to the update (i.e. the original version of ‘fname’ can be found in ‘fname~’).

ta, tb: array(1..3, s) of real
  t2: array(1..5) of integer
end-declarations

initializations from 'a.dat'
  [ta, tb] as 't'
    ! ta = [(1, 'un', 10), (3, 'trois', 30)]
    ! tb = [(1, 'un', 11), (2, 'deux', 22), (3, 'trois', 33)]
  t2
    ! t2 = [10, 0, 0, 30, 40]
  nb_used as "nb used" ! nb_used = 0
end-initializations

nb_used + 1
  ta(2, "quatre") := 1000

initializations to 'a.dat'
  [ta, tb] as 't'
  nb_used as "nb used"
  s
end-initializations
end-model

After the execution of this model, the data file contains:

  ! Example of the use of initialization blocks
  t: [(1 'un') [10 11] (2 'deux') [* 22] (2 'quatre') [1000 *]
      (3 'trois') [30 33]]
  t2: [10 (4) 30 40 ]
  'nb used': 1
  's': ['un' 'deux' 'trois' 'quatre']

Selections

If Statement

The general form of the if statement is:

  if Bool_exp_1
  then Statement_list_1
  [  
    elif Bool_exp_2
    then Statement_list_2
  ]
The selection is executed as follows: if \( \text{Bool\_exp}\_1 \) is true then \( \text{Statement\_list}\_1 \) is executed and the process continues after the end-if instruction. Otherwise, if there are elif statements, they are executed in the same manner as the if instruction itself. If, all boolean expressions evaluated are false and there is an else instruction, then \( \text{Statement\_list}\_E \) are executed; otherwise no statement is executed and the process continues after the end-if keyword.

```plaintext
if c=1 then writeln(’c=1’) 
elif c=2 then writeln(’c=2’) 
else writeln(’c<>1 and c<>2’) 
end-if
```

### Case Statement

The general form of the case statement is:

```plaintext
case Expression\_0 of 
Expression\_1: Statement\_1 
or 
Expression\_1: do Statement\_list\_1 end-do 
[
   Expression\_2: Statement\_2 
or 
   Expression\_2: do Statement\_list\_2 end-do 
...] 
[ else Statement\_list\_E ]
end-case
```

The selection is executed as follows: \( \text{Expression}\_0 \) is evaluated and compared sequentially with each expression of the list \( \text{Expression}\_i \) until a match is found. Then the statement \( \text{Statement}\_i \) (resp. list of statements \( \text{Statement}\_list\_i \)) corresponding to the matching expression is executed and the execution continues after the end-case instruction. If no matching is found and an else statement is present, the list of statements \( \text{Statement}\_list\_E \) is executed, otherwise the execution continues after the end-case instruction. Note that, each of the expression lists \( \text{Expression}\_i \) can be either a scalar, a set or a list of expressions separated by commas. In the last two
cases, the matching succeeds if the expression Expression_0 corresponds to an element of the set or an entry of the list.

case c of
  1     : writeln(‘c=1’)
  2..5  : writeln(‘c in 2..5’)
  6,8,10: writeln(‘c in {6,8,10}’)
  else writein(‘c in {7,9} or c >10 or c <1’)
end-case

Loops

Forall Loop

The general form of the ‘forall’ statement is:

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

The statement Statement (resp. list of statements Statement_list) is repeated for each possible index tuple generated by the iterator list (c.f. “Aggregate Operators”).

forall (i in 1..10,j in 1..10|i<>j) do
  write(‘(’,i,’,’j,’)’)
  if isodd(i*j) then s+={i*j}
end-if
end-do

While Loop

The general form of the ‘while’ statement is:

while (Bool_expr) Statement
or
while (Bool_expr) do Statement_list end-do

The statement Statement (resp. list of statements Statement_list) is repeated as long as the condition Bool_expr is true. If the condition is false at the first evaluation, the while statement is entirely skipped.

i:=1
while(i<=10) do
write(' ',i)
  if isodd(i) then s+=i
end-if
i+=1
end-do

**Repeat Loop**

The general form of the 'repeat' statement is:

```
repeat
  Statement1
  [ Statement2 ...]
until Bool_expr
```

The list of statements enclosed in the instructions *repeat* and *until* is repeated until the condition *Bool_expr* is true. As opposed to the *while* loop, the statement(s) is (are) executed at least once.

```
i:=1
repeat
  write(' ',i)
  if isodd(i) then s+=i
end-if
  i+=1
until i>10
```

**break and next statements**

The statements *break* and *next* are respectively used to interrupt and jump to the next iteration of a loop. The general form of the *break* and *next* statements is:

```
break [n]  
or
next [n]
```

where *n* is an optional integer constant: *n*-1 nested loops are stopped before applying the operation.

```
! in this example only the loop controls are shown
repeat     !1:loop L1
forall (i in S) do !2:loop L2
  while (C3) do !3:loop L3
```
break 3 !4: Stop the 3 loops and continue after line 11
next !5: go to next iteration of L3 (line 3)
next 2 !6: Stop L3 and go to next ‘i’ (line 2)
end-do !7: end of L3
next 2 !8: Stop L2, go to next iteration of L1 (line 11)
break !9: Stop L2 and continue after line 10
end-do !10: end of L2
until C1 !11: end of L1

2.9 Procedures and Functions

It is possible to group sets of statements and declarations in the form of subroutines that, once defined, can be called several times during the execution of the model. There are two kinds of subroutines in Mosel, procedures and functions. Procedures are used in the place of statements (e.g. `writeln("Hi!")`) and functions as part of expressions (because a value is returned, e.g. `round(12.3)`). Procedures and functions may both receive arguments, define local data and call themselves recursively.

**Definition**

Defining a subroutine consists of describing its external properties (i.e. its name and arguments) and the actions to be performed when it is executed (i.e. the statements to perform). The general form of a procedure definition is:

```
procedure name_proc [ (list_of_parms) ]
    Proc_body
end-procedure
```

where `name_proc` is the name of the procedure and `list_of_parms` its list of formal parameters (if any). This list is composed of symbol declarations (c.f. Section 2.6, “The Declaration Block”) separated by commas. The only difference from usual declarations is that no constants or expressions are allowed, including in the indexing list of an array (for instance ‘A=12’ or ‘t1:array(1..4) of real’ are not valid parameter declarations). The body of the procedure is the usual list of statements and declaration blocks except that no procedure or function definition can be included.

```
procedure myproc
    writeln("In myproc")
end-procedure
```
procedure withparams(a:array(r:range) of real, i,j:integer)
    writeln("I received: i="",i," j="",j)
    forall(n in r) writeln("a(",n,")="",a(n))
end-procedure

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

myproc                         ! Call myproc
withparams(mytab,23,67)        ! Call withparams

The definition of a function is very similar to the one of a procedure:

function name_func [(List_of_params)]: Basic_type
    Func_body
end-function

The only difference with a procedure is that the function type must be specified. Mosel supports only functions of basic types (integer, real, boolean and string). Inside the body of a function, a special variable of the type of the function is automatically defined: returned. This variable is used as the return value of the function, it must therefore be assigned a value during the execution of the function.

function multiply_by_3(i:integer):integer
    returned:=i*3
end-function

writeln("3*12=",multiply_by_3(12)) ! Call the function

Formal Parameters: Passing Convention

Formal Parameters of basic types are passed by value and all other types are passed by reference. In practice, when a parameter is passed by value, the subroutine receives a copy of the information so, if the subroutine modifies this parameter, the effective parameter remains unchanged. But if a parameter is passed by reference, the subroutine receives the parameter itself. As a consequence, if the parameter is modified during the process of the subroutine, the effective parameter is also affected.
procedure alter(s:set of integer,i:integer)
  i+=1
  s+=i
end-procedure

gs:={1}
gi:=5
alter(gs,gi)
writeln(gs,gi)          ! displays: {1,6} 5

Local Declarations

Several declaration blocks may be used in a subroutine and all identifiers declared are
local to this subroutine. This means that all of these symbols exist only in the scope of
the subroutine (i.e. between the declaration and the end-procedure or end-
function statement) and all of the resource they use is released once the subroutine
terminates its execution unless they are part of a problem. Decision variables (mpvar)
and active linear constraints (linctr that are not just linear expressions) are
therefore preserved. As a consequence, any decision variables or constraints declared
inside a subroutine are still effective after the termination of the subroutine even if
the symbol used to name the related object is not defined any more.
Note also that a local declaration may hide a global symbol.

declarations         ! global definition
  i,j:integer
end-declarations

procedure myproc
  declarations
    i:string           ! this declaration hides the global symbol
  end-declarations
  i:="a string"      ! local 'i'
  j:=4
  writeln("Inside of myproc, i=",i," j=",j)
end-procedure

i:=45                ! global 'i'
j:=10
myproc
writeln("Outside of myproc, i=",i," j=",j)  ! displays:
Inside of myproc, i=a string j=4
Outside of myproc, i=45 j=4

Overloading

Mosel supports overloading of procedures and functions. One can define the same function several times with different sets of parameters and the compiler decides which subroutine to use depending on the parameter list. This also applies to predefined procedures and functions.

! returns a random number between 1 and a given upper limit
function random(limit:integer):integer
  returned:=round(.5+random*limit) ! use the predefined
  ! 'random' function
end-function

It is important to note that:

- a procedure cannot overload a function and vice versa;
- it is not possible to redefine any identifier; this rule also applies to procedures and functions. A subroutine definition can be used to overload another subroutine only if it differs for at least one parameter. This means, a difference in the type of the return value of a function is not sufficient.

Forward Declaration

During the compilation phase of a source file, only symbols that have been previously declared can be used at any given point. If two procedures call themselves recursively (cross recursion), it is therefore necessary to be able to declare one of the two procedures in advance. Moreover, for the sake of clarity it is sometimes useful to group all procedure and function definitions at the end of the source file. A forward declaration is provided for these uses: it consists of stating only the header of a subroutine that will be defined later. The general form of a forward declaration is:

forward procedure Proc_name [(List_of_params)]
or
forward function Func_name [(List_of_params)]: Basic_type

where the procedure or function Func_name will be defined later in the source file. Note that a forward definition for which no actual definition can be found is considered as an error by Mosel.
2.10 The public qualifier

Once a source file has been compiled, the identifiers used to designate the objects of the model become useless for Mosel. In order to access information after a model has been executed (for instance using the ‘display’ command of the command line interpreter), a table of symbols is saved in the BIM file. If the source is compiled with the strip option (-s), all private symbols are removed from the symbol table – by default all symbols are considered to be private.

The qualifier public can be used in declaration and definition of objects to mark those identifiers (including parameters and subroutines) that must be published in the table of symbols even when the strip option is in use.

    parameters
    public T="default"      ! T is published
    end-parameters

    declarations
    public a,b,c:integer    ! a, b and c are published
    d:real                  ! d is private
    end-declarations

    forward public procedure myproc(i:integer)
    ! 'myproc' is published

2.11 Handling of Input/Output

At the start of the execution of a program/model, two text streams are created automatically: the standard input stream and the standard output stream. The standard output stream is used by the procedures writing text (write, writeln,
The standard input stream is used by the procedures reading text (read, readln, fskipline). These streams are inherited from the environment in which Mosel is being run: usually using an output procedure implies printing something to the console and using an input procedure implies expecting something to be typed by the user.

The procedures fopen and fclose make it possible to associate text files to the input and output streams: in this case the IO functions can be used to read from or write to files. Note that when a file is opened, it is automatically made the active input or output stream (according to its opening status) but the file that was previously assigned to the corresponding stream remains open. It is however possible to switch between different open files using the procedure fselect in combination with the function getfid.

model "test IO"
def_out:=getfid(F_OUTPUT)  ! save file ID of default output
fopen("mylog.txt",F_OUTPUT)! switch output to 'mylog.txt'
my_out:=getfid(F_OUTPUT) ! save ID of current output stream
repeat
  fselect(def_out)   ! select default output...
  write("Text? ")   ! ...to print a message
  text:="'
  readln(text)       ! read a string from the default input
  fselect(my_out)    ! select the file 'mylog.txt'
  writeln(text)      ! write the string into the file
  until text="'
fclose(F_OUTPUT)     ! close current output (=‘mylog.txt’) 
writeln("Finished!")! display message to default output
end-model
3 Predefined Functions and Procedures

This chapter lists in alphabetical order all the predefined functions and procedures included in the Mosel language. Certain functions or procedures take predefined constants as input values or return values that correspond to predefined constants. In every case, these constants are documented with the function or procedure. In addition, Mosel defines a few other useful numerical constants:

- **MAX_INT**  maximum integer number
- **MAX_REAL** maximum real number
- **M_E** base of natural logarithms, e
- **M_PI** value of π
abs

Purpose
Get the absolute value of an integer or real.

Synopsis
function abs(i: integer): integer
function abs(r: real): real

Arguments
i Integer number for which to calculate the absolute value.
r Real number for which to calculate the absolute value.

Return Value
Absolute value of the argument

Related Topics
exp, ln, log, sqrt.
**Purpose**
Get the arctangent of a value.

**Synopsis**
function arctan(r: real): real

**Arguments**
r Real number to which to apply the trigonometric function.

**Return Value**
Arctangent of the argument

**Example**
The following functions compute the arcsine and arccosine of a value:

```mosel
function arcsin(s:real):real
    returned:=arctan(s/(1-s^2))
end-function

function arccos(c:real):real
    returned:=arctan((1-c^2)/c)
end-function
```

**Related Topics**
- cos, sin.
**bittest**

**Purpose**
Test bit settings.

**Synopsis**
```plaintext
function bittest(i: integer, mask: integer): integer
```

**Arguments**
- `i`: Non-negative integer to be tested.
- `mask`: Bit mask.

**Return Value**
Bits selected by the mask.

**Example**
In the following, `i` takes the value 4, `j` the value 5 and `k` the value 8:
```plaintext
i := bittest(12,5)
j := bittest(13,5)
k := bittest(13,10)
```

**Further Information**
This function compares a given number with a bit mask and returns those bits selected by the mask that are set in the number (bit 0 has value 1, bit 1 has value 2, bit 2 has value 4, and so on).
**ceil**

**Purpose**
Round a number to the next largest integer.

**Synopsis**
function ceil(r: real): integer

**Arguments**
r Real number to be rounded.

**Return Value**
Rounded value.

**Example**
In the following, i takes the value 6, j the value -6 and k the value 13:

```plaintext
  i := ceil(5.6)
  j := ceil(-6.7)
  k := ceil(12.3)
```

**Related Topics**
floor, round.
**COS**

**Purpose**
Get the cosine of a value.

**Synopsis**
function cos(r: real): real

**Arguments**
r
Real number to which to apply the trigonometric function.

**Return Value**
Cosine value of the argument.

**Example**
The function tangent can be implemented as follows:

```plaintext
function tangent(x: real): real
  returned := sin(x) / cos(x)
end-function
```

**Related Topics**
arctan, sin.
create

Purpose
Create a decision variable that is part of a previously declared dynamic array.

Synopsis
procedure create(x: mpvar)

Arguments

x Variable to be created.

Example
The following declares a dynamic array of variables, creating only those corresponding to the odd indices. Finally, it defines the linear expression $x(1)+x(3)+x(5)+x(7)$:

```mosel
declarations
  x: dynamic array(1..8) of mpvar
end-declarations

c := sum(i in 1..8| isodd(i)) create(x(i))
```

Further Information
If an array of variables is declared dynamic (or indexed by an empty dynamic set at declaration time) its elements are not created at its declaration. They need to be created subsequently using this procedure.

Related Topics
“Arrays” in Chapter 2, “The Mosel Language”.
exists

Purpose
Check if a given entry in a dynamic array has been created.

Synopsis
function exists(x): boolean

Arguments
x

Array reference (e.g. t(1)).

Return Value
true if the entry exists, false otherwise.

Example
In the following, a dynamic array of decision variables only has its even elements created, which is checked by displaying the existing variables:

declarations
x: dynamic array(1..8) of mpvar
end-declarations

forall(i in 1..8| not isodd(i))
create(x(i))

forall(i in 1..8| exists(x(i))
writeln('x(',i,') exists')

Further Information
If an array is declared dynamic (or indexed by a dynamic set) its elements are not created at its declaration. This function indicates if a given element has been created.

Related Topics
create.
exit

Purpose
Terminate the current program.

Synopsis
procedure exit(code: integer)

Arguments
code Value to be returned by the program.

Further Information
Models exit by default with a value of 0 unless this is changed using exit.
exp

Purpose
Get the natural exponent of a value.

Synopsis
function exp(r: real): real

Arguments
r Real value the function is applied to.

Return Value
Natural exponent of the argument.

Related Topics
abs, ln, log, sqrt.
exportprob

Purpose
Export a problem to a file.

Synopsis
procedure exportprob(options: integer, filename: string,
obj: linctr)

Arguments
options  File format options:
EP_MIN    LP format, minimization (default);
EP_MAX    LP format, maximization;
EP_MPS    MPS format;
EP_STRIP  use scrambled names.
Several options may be combined using ‘+’.
filename  Name of the output file. If the empty string "" is given, output
is printed to the standard output (the screen).
obj  Objective function constraint.

Further Information
1. If the given filename has no extension, Mosel appends .lp to it for LP format
files and .mat for MPS format.
2. When exporting matrices in MPS format any possibly specified lower bounds
on semi-continuous or semi-continuous integer variables are lost. LP format
matrices maintain the complete information.


**fclose**

**Purpose**
Close the active input or output stream.

**Synopsis**
```
procedure fclose(stream: integer)
```

**Arguments**
- `stream`: The stream to close:
  - F_INPUT  input stream;
  - F_OUTPUT output stream.

**Further Information**
This procedure flushes pending data (for output stream), and then closes the file that is currently associated with the given stream. The file preceding the closed file (in the order of opening) is then assigned to the corresponding stream. A file that is closed with this procedure must previously have been opened with `fopen`. This function has no effect if the corresponding stream is not associated with any explicitly opened file (i.e., it is not possible to close the default input or output stream). All open streams are automatically closed when the program terminates.

**Related Topics**
- `fopen`, `fselect`, `getfid`, `iseof`, `fflush`. 
**fflush**

**Purpose**
Force the operating system to write all buffered data.

**Synopsis**

```plaintext
procedure fflush
```

**Further Information**
This procedure forces a write of all buffered data of the default output stream. `fflush` is automatically called when the stream is closed either with `fclose` or when the program terminates.

**Related Topics**
`fopen`, `fclose`.
**finalize**

**Purpose**
Finalize the definition of a set.

**Synopsis**
procedure finalize(s: set)

**Arguments**

- s Dynamic set.

**Example**
In the following, an indexing set is defined, on which depends a dynamic array of decision variables. The set is subsequently defined to have three elements and is finalized. A static array is then defined:

```plaintext
declarations
  Set1: set of string
  x: array(Set1) of mpvar  ! x is dynamic
end-declarations

Set1 := {"first", "second", "fifth"}
finalize(Set1)

declarations
  y: array(Set1) of mpvar  ! y is static
end-declarations
```

**Further Information**
This procedure finalizes the definition of a set, that is, it turns a dynamic set into a constant set consisting of the elements that are currently in the set. All subsequently declared arrays that are indexed by this set will be created as static (i.e. of fixed size). Any arrays indexed by this set that have been declared prior to finalizing the set retain the status dynamic but their set of elements cannot be modified any more.
floor

Purpose
Round a number to the next smallest integer.

Synopsis
function floor(r: real): integer

Arguments
r Real number to be rounded.

Return Value
Rounded value.

Example
In the following, i takes the value 5, j the value –7 and k the value 12:
i := floor(5.6)
j := floor(-6.7)
k := floor(12.3)

Related Topics
cell, round.
fopen

Purpose
Open a file and make it the active input or output stream.

Synopsis
procedure fopen(f: string, mode: integer)

Arguments
f The name of the file to be opened.
mode Open mode:
- F_INPUT open for reading;
- F_OUTPUT empty the file and open it for writing;
- F_APPEND open for writing, appending new data to the end of the file.

Further Information
1. This procedure opens a file for reading or writing. If the operation succeeds, depending on the opening mode, the file becomes the active (default) input or output stream. The procedures write and writeln are used to write data to the default output stream and the functions read, readln and fskipline are used to read data from the default input stream.

2. The behaviour of this function in case of an IO error (i.e. the file cannot be opened) is directed by the control parameter IOCTRL: if the value of this parameter is false (default value), the interpreter stops. Otherwise, the interpreter ignores the error and continues. The error status of an IO operation is stored in the control parameter IOSTATUS which is 0 when the last operation has been executed successfully. Note that this parameter is automatically reset once its value has been read using the function getparam. The behaviour of IO operations after an unhandled error is not defined.

Related Topics
fclose, fselect, getfid.
**fselect**

**Purpose**
Select the active input or output stream.

**Synopsis**

```plaintext
procedure fselect(stream: integer)
```

**Arguments**

- `stream` The stream number.

**Related Controls**
None.

**Example**
The following saves the file ID of the default output, before switching output to the file `mylog.txt`. Subsequently, the file ID of the current output stream is saved and the default output is again selected.

```plaintext
def_out := getfid(F_OUTPUT)
fopen("mylog.txt",F_OUTPUT)
...
my_out:=getfid(F_OUTPUT)
fselect(def_out)
```

**Further Information**
This procedure selects the given stream as the active input or output stream. The stream concerned is designated by the opening status of the given stream (that is, if the given stream has been opened for reading, it will be assigned to the default input stream). The stream number can be obtained with the function `getfid`.

**Related Topics**
`fclose`, `fopen`, `getfid`.
fskipline

Purpose
Advance in the default input stream as long as comment lines are found.

Synopsis
procedure fskipline(filter: string)

Arguments
filter List of comment signs.

Example
In the following, the first statement skips all lines beginning with either ‘#’ or ‘!’.

fskipline("!#")
fskipline("\n")

Further Information
This procedure advances in the input stream using the given list of comment
signs as a filter. Each character of the given string is considered to be a
symbol that marks the beginning of a comment line. Note that the character
‘\n’ designates lines starting with nothing, that is, empty lines. During the
parsing, spaces and tabulations are ignored.

Related Topics
read, readln.
getact

Purpose
Get the activity value of a constraint.

Synopsis
function getact(c: linctr): real

Arguments
c A linear constraint.

Return Value
Activity value or 0.

Further Information
This function returns the activity value of a constraint if the problem has been solved successfully, otherwise 0 is returned.

Related Topics
getdual, getslack, getsol.
getcoeff

Purpose
Get the coefficient of a given variable in a constraint.

Synopsis
function getcoeff(c: linctr): real
function getcoeff(c: linctr, x: mpvar): real

Arguments
c A linear constraint.
x A decision variable.

Return Value
Coefficient of the variable or the constant term

Example
In this example, a single constraint with three variables is defined. The calls to getcoeff result in r taking the value -1 and s taking the value -12.

declarations
   x, y, z: mpvar
end-declarations

c := 4*x + y - z <= 12
r := getcoeff(c,z)
s := getcoeff(c)

Further Information
This function returns the coefficient of a given variable in a constraint, or if no variable is given, the constant term (= -RHS) of the constraint. The returned values correspond to a normalised constraint representation with all variable and constant terms on the left side of the (in)equality sign.

Related Topics
getvars, setcoeff.
**getdual**

**Purpose**
Get the dual value of a constraint.

**Synopsis**
function getdual(c: linctr): real

**Arguments**
c A linear constraint.

**Return Value**
Dual value or 0.

**Further Information**
This function returns the dual value of a constraint if the problem has been solved successfully, otherwise 0 is returned.

**Related Topics**
getrcost, getslack, getsol.
**getfid**

**Purpose**
Get the stream number of the active input or output stream.

**Synopsis**
function getfid(stream: integer): integer

**Arguments**

stream The stream to query:
- F_INPUT input stream;
- F_OUTPUT output stream.

**Return Value**
Stream number.

**Further Information**
The returned value can be used as parameter for the function fselect.

**Related Topics**
fselect.
**getfirst**

**Purpose**
Get the first element of a range set.

**Synopsis**
```plaintext
function getfirst(r: range): integer
```

**Arguments**
r  A range set.

**Return Value**
The first element of the set.

**Example**
In this example, the range set `r` is defined before its first and last elements are retrieved and displayed:
```plaintext
declarations
  r = 2..8
end-declarations
...
writeln("First element of r: ", getfirst(r),"\nLast element of r: ",getlast(r))
```

**Further Information**
This function returns the first element of a range set. If the range is empty, this function returns 0.

**Related Topics**
`getlast`. 
getlast

Purpose
Get the last element of a range set.

Synopsis
function getlast(r: range): integer

Arguments
r A range set.

Return Value
The last element of the set

Example
In this example, the range set r is defined before its first and last elements are retrieved and displayed:

declarations
r = 2..8
end-declarations
...
writeln("First element of r: ", getfirst(r),"\nLast element of r: ", getlast(r))

Further Information
This function returns the last element of a range set. If the range is empty, this function returns -1.

Related Topics
getfirst.
getobjval

Purpose
Get the objective function value.

Synopsis
function getobjval: real

Return Value
Objective function value or 0

Further Information
This function returns the objective function value if the problem has been solved successfully, otherwise 0 is returned. If integer feasible solution(s) have been found, the value of the best is returned, otherwise the value of the last LP solved.

Related Topics
getsol.
**getparam**

**Purpose**
Get the current value of a control parameter.

**Synopsis**
```
function getparam(name: string):
    integer|string|real|boolean
```

**Arguments**
- **name**
  Name of the control parameter whose value is to be returned (case insensitive).

**Return Value**
Current setting of the control parameter

**Example**
In the following, the automatic IO error checking of Mosel is disabled and then we try to open a file and check the error status to make sure the operation succeeded:
```
setparam("IOCTRL",false)
fopen('myfile',F_INPUT)
if getparam("IOSTATUS")<>0 then
    writeln("fopen failed - aborting")
    exit(1)
end-if
```

**Further Information**
1. Parameters whose values may be returned by this command include the settings of Mosel as well as those of any loaded module. The module may be specified by prefixing the parameter name with the name of the module (e.g. `mmxprs.XPRS_verbose`). The type of the return value corresponds to the type of the parameter.
2. This function can be applied only to control parameters whose value can be accessed.
3. The following control parameters are supported by Mosel:
   - `realfmt`  default C printing format for real numbers (string)
   - `zerotol`  zero tolerance in comparisons between reals (real)
iocr1    the interpreter ignores IO errors (Boolean)
iostatus status of the last IO operation (integer)
nbread  number of items recognised by the last read procedure or read
        in by the last initializations block (integer)

Related Topics
setparam.
getrcost

Purpose
Get the reduced cost value of a decision variable.

Synopsis
function getrcost(v: mpvar): real

Arguments

v A decision variable.

Return Value
Reduced cost value or 0.

Further Information
This function returns the reduced cost value of a decision variable if the problem has been solved successfully, otherwise 0 is returned.

Related Topics
getslack, getsol, getdual.
**getsize**

**Purpose**
Get the size of an array, set or a string.

**Synopsis**

```plaintext
function getsize(a: array): integer
function getsize(s: set): integer
function getsize(t: string): integer
```

**Arguments**

- `a` An array.
- `s` A set.
- `t` A string.

**Return Value**
Number of effective cells for an array, number of elements for a set, number of characters for a string.

**Example**
In the following, a dynamic array is declared holding eight elements, of which only two are actually defined. Calling `getsize` on this array returns 2 rather than 8.

```plaintext
declarations
    a: dynamic array(1..8) of real
end-declarations

    a(1) := 4
    a(5) := 7.2
    l := getsize(a)
```

**Further Information**
This function returns the size of an array or a set, that is the number of cells or elements. In the case of a dynamic array that has been declared with a maximal range this number may be smaller than the size of the range, but it cannot exceed it. When used with a string, this function returns the length of the strings (i.e. the number of characters it contains).
**getslack**

**Purpose**
Returns the slack value of a constraint if the problem has been solved successfully and the constraint is contained in the problem, or 0 otherwise.

**Synopsis**
function getslack(c: linctr): real

**Arguments**
c A linear constraint.

**Return Value**
Slack value or 0.

**Further Information**
This function returns the slack value of a constraint if the problem has been solved successfully, otherwise 0 is returned.

**Related Topics**
getdual, getrcost, getsol.
getsol

Purpose
Get the solution value of a variable or a linear expression (constraint).

Synopsis
function getsol(v: mpvar): real
function getsol(c: linctr): real

Arguments
v  A decision variable.
c  A linear constraint.

Return Value
Solution value or 0.

Further Information
This function returns the (primal) solution value of a variable if the problem has been solved successfully and the variable is contained in the problem (otherwise 0). If used with a constraint, it returns the evaluation of the corresponding linear expression using the current solution.

Related Topics
getdual, getrcost, getobjval.
gettype

Purpose
Get the type of a linear constraint.

Synopsis
function gettype(c: linctr): integer

Arguments
c A linear constraint.

Return Value
Constraint type. Values applicable to any type of linear constraint are:

- **CT_EQ** equality, ’ = ’
- **CT_GEQ** greater than or equal to, ‘ ≥ ’
- **CT_LEQ** less than or equal to, ‘ ≤ ’
- **CT_UNB** nonbinding constraint
- **CT_SOS1** special ordered set of type 1
- **CT_SOS2** special ordered set of type 2

Values applicable for unary constraints are:

- **CT_CONT** continuous
- **CT_INT** integer
- **CT_BIN** binary
- **CT_PINT** partial integer
- **CT_SEC** semi-continuous
- **CT_SINT** semi-continuous integer

Related Topics
settype.
getvars

**Purpose**
Get the set of variables of a constraint.

**Synopsis**
```plaintext
procedure getvars(c: linctr, s: set of mpvar)
```

**Arguments**
- `c`  A linear constraint.
- `s`  Where the set of decision variables is returned.

**Example**
The following returns the set of variables in a linear constraint to the set variable `vset`, and then loops through them to find their solution values:
```plaintext
declarations
  c: linctr
  vset: set of mpvar
end-declarations

getvars(c,vset)
forall(x in vset) writeln(getsol(x))
```

**Further Information**
This procedure returns in the parameter `s` the set of variables of a constraint. Note that this procedure replaces the content of the set.
iseof

Purpose
Test whether the end of the default input stream has been reached.

Synopsis
function iseof: boolean

Return Value
true if the end of the default input stream has been reached, false otherwise

Example
The following opens a datafile of integers, reads one from each line and prints it to the console until the end of the file is reached:

declarations
d: integer
end-declarations
...
fopen("datafile.dat",F_INPUT)
while(not iseof) do
  readln(d)
  writeln(d)
end-do
fclose(F_INPUT)

Related Topics
fclose, fopen.
ishidden

Purpose
Test whether a constraint is hidden.

Synopsis
function ishidden(c: linctr): boolean

Arguments
  c  A linear constraint.

Return Value
ture if the constraint is hidden, false otherwise.

Further Information
At its creation a constraint is added to the current problem, but using the
function sethidden it may be hidden. This means that the constraint will
not be contained in the problem that is solved by the optimizer but it is not
deleted from the definition of the problem in Mosel.

Related Topics
sethidden.
**isodd**

**Purpose**
Test whether an integer is odd.

**Synopsis**
function isodd(i: integer): boolean

**Arguments**
i An integer number.

**Return Value**
true if the given integer is odd, false if it is even
**ln**

**Purpose**
Get the natural logarithm of a value.

**Synopsis**
```plaintext
function ln(r: real): real
```

**Arguments**
- `r` Real value the function is applied to. This must be positive.

**Return Value**
Natural logarithm of the argument.

**Example**
The following example provides a function for calculating logarithms to any (positive) base:
```plaintext
function logn(base,number: real): real
if(number > 0 and base > 0) then
    returned := ln(number)/ln(base)
else
    exit(1)
end-if
end-function
```

**Related Topics**
- exp, log, sqrt.
**log**

**Purpose**
Get the base 10 logarithm of a value.

**Synopsis**
```plaintext
function log(r: real): real
```

**Arguments**
- `r` Real value the function is applied to. This must be positive.

**Return Value**
Base 10 logarithm of the argument.

**Related Topics**
- `exp`, `ln`, `sqrt`. 
**makesos**

**Purpose**
Creates a special ordered set (SOS) using a set of decision variables and a linear constraint.

**Synopsis**

```
procedure makesos1(cs: linctr, s: set of mpvar, c: linctr)
procedure makesos1(s: set of mpvar, c: linctr)
procedure makesos2(cs: linctr, s: set of mpvar, c: linctr)
procedure makesos2(s: set of mpvar, c: linctr)
```

**Arguments**

- **cs** A linear constraint.
- **s** A set of decision variables.
- **c** A linear constraint.

**Example**

The following generates the SOS1 set `mysos` based on the linear constraint `rr`. The resulting set contains the variables `x`, `y` and `z` with the weights 0, 2 and 4.

```
declarations
  x,y,z: mpvar
  rr,mysos: linctr
end-declarations

rr:=2*y+4*z
makesos1(mysos,(x,y,z),rr)
```

**Further Information**
These procedures generate a SOS set containing the decision variables of the set `s` with the coefficients of the linear constraint `c`. The resulting set is assigned to `cs` if it is provided. Note that these procedures simplify the generation of SOS sets with weights of value 0.
**maxlist**

**Purpose**
Get the maximum value of a list of integers or reals.

**Synopsis**
```plaintext
function maxlist(i1: integer, i2: integer [, i3: integer ...]): integer
function maxlist(r1: real, r2: real [, r3: real ...]): real
```

**Arguments**
- `i1, i2, ...` List of integer numbers.
- `r1, r2, ...` List of real numbers.

**Return Value**
Largest value in the given list.

**Example**
In the following `r` is assigned the value 7 by `maxlist`:
```plaintext
r := maxlist(-1, 4.5, 2, 7, -0.3)
```

**Further Information**
The returned type corresponds to the type of the input.

**Related Topics**
- `minlist`
minlist

Purpose
Get the minimum value of a list of integers or reals.

Synopsis
function minlist(i1: integer, i2: integer [,i3: integer...]): integer
function minlist(r1: real, r2: real [,r3: real...]): real

Arguments
i1, i2, ... List of integer numbers.
r1, r2, ... List of real numbers.

Return Value
Smallest value in the given list.

Example
In the following, r is assigned the value -1 by minlist:
r := minlist(-1, 4.5, 2, 7, -0.3)

Further Information
The returned type corresponds to the type of the input.

Related Topics
maxlist.
random

Purpose
Generate a random number.

Synopsis
function random: real

Return Value
A randomly generated number in the range [0,1).

Example
In the following, i is assigned a random integer value between 1 and 10:

\[
i := \text{integer} \left( \text{round} \left( (10 \times \text{random}) + 0.5 \right) \right)
\]

Further Information
Each model uses its own generator which is randomly initialized when the model execution starts. The sequence may also be reset using procedure setrandseed.

Related Topics
setrandseed.
**read, readln**

**Purpose**
Read in formatted data from the active input stream.

**Synopsis**

```
procedure read(e1: expr [,e2: expr...])
procedure readln
procedure readln(e1: expr [,e2: expr...])
```

**Arguments**

\(e1,e2,...\) Expression, or list of expressions, of basic type.

**Example**

The following reads (possibly split over several lines) 12 45 word, followed by \(\text{toto}(12\text{ and }45)=\text{word}\):

```
declarations
  i, j: integer
  s: string
  ts: array(range,range) of string
end-declarations

read(i,j,s)
readln("\text{toto}("i,"\text{and}"j,"\text{=}","ts(i,j))
```

**Further Information**

1. These procedures assign the data read from the active input stream to the given symbols or try to match the given expressions with what is read from the input stream. If \(ei\) is a symbol that can be assigned a value, the procedure tries to recognise from the input stream a constant of the required type and, if successful, assigns the resulting value to \(ei\). If \(ei\) is a constant or a symbol that cannot be reassigned, the procedure tries to read in a constant of the required value and succeeds if the resulting value corresponds to \(ei\). These procedures do not fail but set the control parameter \(\text{NBREAD}\) to the number of items actually recognised.

2. Note that the read procedures are based on the lexical analyser of Mosel: items are separated by spaces and a string that contains spaces must be quoted using either single or double quotes (the quotes are automatically removed once the string has been identified).
3. The procedure \texttt{readln} expects all the items to be recognised to be contained in single line. The function \texttt{read} ignores changes of line. If the procedure \texttt{readln} is used without parameters it skips to the end of the current line.

\textbf{Related Topics}

\texttt{write}, \texttt{writeln}.
round

Purpose
Round a number to the nearest integer.

Synopsis
function round(r: real): integer

Arguments
r Real number to be rounded.

Return Value
Rounded value.

Example
In the following, i takes the value 6, j the value −7 and k the value 12:
i := round(5.6)
j := round(-6.7)
k := round(12.3)

Related Topics
cell, floor.
**setcoeff**

**Purpose**
Set the coefficient of a variable or the constant term in a constraint.

**Synopsis**

```plaintext
procedure setcoeff(c: linctr, x: mpvar, r: real)
procedure setcoeff(c: linctr, r: real)
```

**Arguments**

- `c`  A linear constraint.
- `x`  A decision variable.
- `r`  Coefficient or constant term.

**Example**

The following declares a constraint `c` and then changes some of its terms:

```plaintext
declarations
x,y,z: mpvar
end-declarations

c := 4*x + y - z <= 12

setcoeff(c,y,2)
setcoeff(c,8.1)
```

The constraint is now `4*x + 2*y - z <= -8.1`.

**Further Information**

If a variable is given then this procedure sets the coefficient of this variable in the constraint to the given value. Otherwise, it sets the constant term of the constraint.

**Related Topics**

- `getcoeff`.
sethidden

Purpose
Hide or unhide a constraint.

Synopsis
procedure sethidden(c: linctr, b: boolean)

Arguments
- c A linear constraint.
- b Constraint status:
  - true hide the constraint;
  - false unhide the constraint.

Example
The following defines a constraint and then sets it as hidden:

declarations
    x,y,z: mpvar
end-declarations

c := 4*x + y - z <= 12
sethidden(c, true)

Further Information
At its creation a constraint is added to the current problem, but using this procedure it may be hidden. This means that the constraint will not be contained in the problem that is solved by the optimizer but it is not deleted from the definition of the problem in Mosel. Function ishidden can be used to test the current status of a constraint.

Related Topics
ishidden.
setparam

**Purpose**
Set the value of a control parameter.

**Synopsis**
```
procedure setparam(name: string,
val: integer|string|real|boolean)
```

**Arguments**
- **name** Name of the control parameter whose value is to be set (case insensitive).
- **val** New value for the control parameter.

**Example**
See example of function `getparam`.

**Further Information**
1. Control parameters include the settings of Mosel as well as those of any loaded module. The module may be specified by prefixing the parameter name with the name of the module (e.g. `'mmxprs.XPRS_loadnames'`). The type of the value must correspond to the type expected by the parameter.
2. This procedure can be applied only to control parameters which value can be modified.
3. The following control parameters, supported by Mosel, can be altered with this procedure:
   - **realfmt** default C printing format for real numbers (string, default: "%g")
   - **zerotol** zero tolerance when comparing reals (real, default: 1e-6)
   - **ioctrl** the interpreter ignores IO errors (Boolean, default: false)

**Related Topics**
`getparam`.
setrandseed

Purpose
Initialize the random number generator.

Synopsis
procedure setrandseed(s: integer)

Arguments
\( s \)
Seed value.

Further Information
This procedure sets its argument as the seed for a new sequence of pseudo-random numbers to be returned by the function random.

Related Topics
random.
settype

Purpose
Set the type of a constraint.

Synopsis
procedure settype(c: linctr, type: integer)

Arguments
c A linear constraint.
type Constraint type.

Further Information
The type (type) of a linear constraint may be set to one of:
- CT_EQ equality, ‘=’
- CT_GEQ greater than or equal to, ‘≥’
- CT_LEQ less than or equal to, ‘≤’
- CT_UNB nonbinding constraint
- CT_SOS1 special ordered set of type 1
- CT_SOS2 special ordered set of type 2

Values applicable for unary constraints only are:
- CT_CONT continuous
- CT_INT integer
- CT_BIN binary
- CT_PINT partial integer
- CT_SEC semi-continuous
- CT_SINT semi-continuous integer

Related Topics
gertype.
**sin**

**Purpose**
Get the sine of a value.

**Synopsis**

```
function sin(r:real): real
```

**Arguments**

- `r` Real number to which to apply the trigonometric function.

**Return Value**
Sine value of the argument.

**Related Topics**
- `arctan`, `cos`
**sqrt**

**Purpose**
Get the positive square root of a value.

**Synopsis**
function sqrt(r: real): real

**Arguments**
- r Real value to which the function is applied. This must be non-negative.

**Return Value**
Square root of the argument.

**Related Topics**
exp, ln, log.
strfmt

Purpose
Creates a formatted string from a string or a number.

Synopsis
function strfmt(str: string, len: integer): string
function strfmt(i: integer, len: integer): string
function strfmt(r: real, len: integer): string
function strfmt(r: real, len: integer, dec: integer): string

Arguments
str        String to be formatted.
i          Integer to be formatted.
r          Real to be formatted.
len        Reserved length (may be exceeded if given string is longer, in which case the string is always left justified):
            <0  left justified within reserved space;
            >0  right justified within reserved space;
            0   use defaults.
dec        Number of digits after the decimal point.

Return Value
Formatted string.

Example
The following:
writeln("text1", strfmt("text2",8), "text3")
writeln("text1", strfmt("text2",-8), "text3")
r := 789.123456
writeln(strfmt(r,0)," ", strfmt(r,4,2), strfmt(r,8,0))

outputs:
text1 text2 text3
text1 text2 text3
789.123 789.12   789
Further Information
1. This function creates a formatted string from a string or an integer or real number. It can be used at any place where strings may be used. Its most likely use is for generating printed output (in combination with write and writeln).

2. If the resulting string is longer than the reserved space it is not cut but printed in its entirety, overflowing the reserved space to the right.

Related Topics
write, writeln.
**substr**

**Purpose**
Get a substring of a string.

**Synopsis**

```haskell
function substr(str: string, i1: integer, i2: integer): string
```

**Arguments**

- **str** String.
- **i1** Starting position of the substring.
- **i2** End position of the substring.

**Return Value**
Substring of the given string.

**Example**

```haskell
write(substr("Example text",3,10))
```

outputs the text: `ample te`.

**Further Information**

This function returns the substring from the \( i1 \)th to the \( i2 \)th character of a given string (the counting starts from 1). This function returns an empty string if the bounds are not compatible with the string (e.g. starting position larger than the length of the string) or inconsistent (e.g. starting position after end position).
write, writeln

**Purpose**
Send an expression or list of expressions to the active output stream.

**Synopsis**
```
procedure write(e1: expr [,e2: expr...])
procedure writeln
procedure writeln(e1: expr [,e2: expr...])
```

**Arguments**
e1, e2,... Expression, or list of expressions.

**Example**
The following defines and outputs a set Set1, prints a blank line and then outputs 'A real:7.12, a Boolean:true':
```
Set1 := {"first", "second", "fifth"}
write(Set1)
writeln
b := true
writeln("A real:", strfmt(7.1234, 4, 2),
"", a Boolean:",b)
```

**Further Information**
These procedures write the given expression or list of expressions to the active output stream. The procedure writeln adds the return character to the end of the output. Numbers may be formatted using function strfmt. Basic types are printed "as is". For elementary but non-basic types (linctr, mpvar) only the address is printed. If the expression is a set or array, all its elements are printed.

**Related Topics**
read, readln, strfmt.
4 mmetc

This compatibility module just defines the diskdata procedure required to use data files formatted for mp-model from Mosel and provides a commercial discounting function. To use this module, the following line must be included in the header of the Mosel model file:

uses 'mmetc'

4.1 Procedures and Functions

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>Pg</th>
</tr>
</thead>
<tbody>
<tr>
<td>disc</td>
<td>Get annual discount</td>
<td>96</td>
</tr>
<tr>
<td>diskdata</td>
<td>Read in or write an array or set of strings from/to a file</td>
<td>97</td>
</tr>
</tbody>
</table>
**disc**

**Purpose**
Get annual discount \((1/(1+a)^{t-1})\)

**Synopsis**

```plaintext
function disc(a: real, t: real)
```

**Arguments**

- \(a\) Discount factor, it must be greater than -1
- \(t\) Time

**Return Value**
Annual discount value.
**diskdata**

**Purpose**
Read or write an array or set of strings from/to a file

**Synopsis**

```plaintext
procedure diskdata(fmt: integer, f: string, a: array)
procedure diskdata(fmt: integer, f: string, s: set of string)
```

**Arguments**

- **fmt** Format options
  - ETC_DENSE dense data format;
  - ETC_SPARSE sparse data format;
  - ETC_SGLQ strings quoted with single quotes;
  - ETC_NOQ strings are not quoted in the file;
  - ETC_OUT write to file;
  - ETC_APPEND append output to the end of the file;
  - ETC_TRANS tables are transposed;
  - ETC_IN read from file (default);
  - ETC_NOZEROS skip zero values.
  Several options may be combined using `+`

- **f** File name
- **a** Array with elements of basic type
- **s** Set of strings

**Example**
The following reads the array `ar1` in sparse format from the file ‘in.dat’ then save `ar1` and `ar2` to the file ‘out.dat’ (in sparse format) and finally appends the contents of the set `Set1` to the file ‘out.dat’:

```plaintext
declarations
Set1: set of string
R: range
ar1, ar2: array(Set1, R) of real
end-declarations

diskdata(ETC_SPARSE, "in.dat", ar1)
diskdata(ETC_OUT, "out.dat", [ar1, ar2])
diskdata(ETC_OUT+ETC_APPEND, "out.dat", Set1)
```
**Further Information**
This procedure reads data from a file or writes to a file, depending on the parameter settings. The file format used is compatible with the command DISKDATA of the modeler mp-model.
5 mmive

The mmive module is used by the Xpress-MP Integrated Visual Environment Xpress-IVE to extend its graphical capabilities. This module supports a set procedures which allow users to display graphs of functions, diagrams, networks, various shapes etc.. To use this module the following line must be included in the header of the Mosel model file:

```
uses 'mmive'
```

Note that this module can be used only from Xpress-IVE (i.e. it is not possible to compile or run a model using it from Mosel Console or the Mosel libraries). The graphs produced by these functions will appear when selecting the ‘User graph’ tab of the Run Pane in Xpress-IVE.

5.1 Procedures and Functions

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>Pg</th>
</tr>
</thead>
<tbody>
<tr>
<td>IVE_RGB</td>
<td>Computes a color based on red, green, blue</td>
<td>100</td>
</tr>
<tr>
<td>IVEaddplot</td>
<td>Inserts a new plot on the user graph</td>
<td>101</td>
</tr>
<tr>
<td>IVEdrawarrow</td>
<td>Draws an arrow from one point to another</td>
<td>102</td>
</tr>
<tr>
<td>IVEdrawlabel</td>
<td>Writes a text label at given coordinates</td>
<td>103</td>
</tr>
<tr>
<td>IVEdrawline</td>
<td>Connects two points using a line</td>
<td>104</td>
</tr>
<tr>
<td>IVEdrawpoint</td>
<td>Draws a square indicating a point</td>
<td>105</td>
</tr>
<tr>
<td>IVEerase</td>
<td>Removes all plots and clears the user graph</td>
<td>106</td>
</tr>
<tr>
<td>IVEpause</td>
<td>Suspend the execution of Mosel until user continues</td>
<td>107</td>
</tr>
<tr>
<td>IVEzoom</td>
<td>Scales the chart between two given points</td>
<td>108</td>
</tr>
</tbody>
</table>

The procedures IVEaddtograph and IVEinitgraph are deprecated and are still included in mmive for backward compatibility. They will be removed in future releases.
IVE_RGB

Purpose
Compute a composite color by combining amounts of red, green and blue.

Synopsis
function IVE_RGB(red: integer, green: integer, blue: integer): integer

Arguments
red Amount of red (between 0 and 255).
green Amount of green (between 0 and 255).
blue Amount of blue (between 0 and 255).

Return Value
The composite color.

Example
The following mixes red with green and stores the result in a variable:
declarations
 a_color: integer
end-declarations

 a_color:=IVE_RGB(255,255,0)

Further Information
If the color component values are out of range, mmive will produce a
warning and return 0 (black).
**IVEaddplot**

**Purpose**
Inserts a new plot on the user graph.

**Synopsis**

```plaintext
function IVEaddplot(name: string, color: integer): integer
```

**Arguments**
- `name`: A string representing the name of the plot which will appear in the legend.
- `color`: An integer representing a color obtained using IVE_RGB or one of the predefined constants:
  - IVE_BLACK
  - IVE_BLUE
  - IVE_CYAN
  - IVE_GREEN
  - IVE_MAGENTA
  - IVE_RED
  - IVE_WHITE
  - IVE_YELLOW.

**Return Value**
An integer representing a handle to this plot. The handle should be stored for later use by the other graphing functions.

**Example**
The following adds two plots to the user graph:

```plaintext
declarations
  plot1, plot2: integer
end-declarations

plot1:=IVEaddplot("sine", IVE_RED)
plot2:=IVEaddplot("random numbers", IVE_GREEN)
```

**Further Information**
1. A plot is identified by its name and can be shown or hidden using its corresponding legend checkbox. A plot controls a virtually unlimited number of points, lines, arrows and labels which were added to it.
2. The maximum number of distinct plots is currently limited to 20. However, each plot can contain an unlimited number of points, lines, arrows and labels.
**IVEdrawarrow**

**Purpose**
Add an arrow to an existing plot. The arrow connects the two points whose coordinates are given as parameters, pointing to the second one.

**Synopsis**
```pascal
procedure IVEdrawarrow(handle: integer, x1: real, y1: real,
                      x2: real, y2: real)
```

**Arguments**
- **handle** The number returned by IVEaddplot.
- **x1** The x coordinate of the first point.
- **y1** The y coordinate of the first point.
- **x2** The x coordinate of the second point.
- **y2** The y coordinate of the second point.

**Example**
The following adds two arrows to a plot named "thetime." The arrows suggest three o'clock:
```pascal
declarations
  arrows: integer
end-declarations

arrows:=IVEaddplot("thetime", IVE_BLACK)
IVEdrawarrow(arrows,0,0,0,5)
IVEdrawarrow(arrows,0,0,4.5,0)
IVEzoom(-5,-6,5,6)
```
**IVEdrawlabel**

**Purpose**
Add a text box to an existing plot. The box will be centered horizontally just above the point given.

**Synopsis**
```
procedure IVEdrawlabel(handle: integer, x: real, y: real,
                        text: string)
```

**Arguments**
- **handle**: The number returned by IVEaddplot.
- **x**: The x coordinate of the point.
- **y**: The y coordinate of the point.
- **text**: The text that will be displayed at the given point.

**Example**
This code complements the time graph with a dial:
```
...!
!this should complement the example for
!IVEdrawarrow

forall (i in 1..12)
  IVEdrawlabel(arrows,
               4.8*cos(1.57-6.28*i/12),5*sin(1.57-6.28*i/12),""+i)
```
**IVEdrawline**

**Purpose**
Add a line to an existing plot. The line connects the two points whose coordinates are given as parameters.

**Synopsis**
```
procedure IVEdrawline(handle: integer, x1: real, y1: real,
                        x2: real, y2: real)
```

**Arguments**
- **handle**: The number returned by IVEaddplot.
- **x1**: The x coordinate of the first point.
- **y1**: The y coordinate of the first point.
- **x2**: The x coordinate of the second point.
- **y2**: The y coordinate of the second point.

**Example**
The following code draws a square, given the correct aspect ratio of the user graph.

```
declarations
    square: integer
end-declarations

    square:=IVEaddplot("square", IVE_YELLOW)
    IVEdrawline(square,-2,-2,-2,2)
    IVEdrawline(square,-2,2,2,2)
    IVEdrawline(square,2,2,2,-2)
    IVEdrawline(square,2,-2,-2,-2)
    IVEzoom(-5,-5,5,5)
```
**IVEdrawpoint**

**Purpose**
Add a small square to mark a point at the given coordinates.

**Synopsis**
```plaintext
procedure IVEdrawpoint(handle: integer, x: real, y: real)
```

**Arguments**
- **handle**: The number returned by IVEaddplot.
- **x**: The x coordinate of the point.
- **y**: The y coordinate of the point.

**Example**
This code plots 100 random points:
```plaintext
declarations
    cloud: integer
end-declarations

cloud:=IVEaddplot("random points",IVE_YELLOW)
IVEzoom(-5,-5,5,5)
forall(i in 1..100)
    IVEdrawpoint(cloud,-2+4*random,-2+4*random)
```
IVEerase

Purpose
Remove all plots and reset the user graph.

Synopsis
procedure IVEerase

Further Information
This procedure can be used together with IVEpause to explore a number of different user graphs during the execution of a Mosel model.

Related Topics
IVEpause
IVEpause

Purpose
Suspends the execution of a Mosel model at the line where the call occurs.

Synopsis
procedure IVEpause(message: string)

Arguments
message
The message will be displayed at the top of the Run Pane in Xpress-IVE.

Further Information
While the run is interrupted, the Xpress-IVE entity tree and other progress graphs can be inspected. This allows precise debugging of Mosel model programs. To continue, click on the Pause button on the toolbar or select the Pause option in the Build menu.
IVEzoom

Purpose
Scales the user graph.

Synopsis
procedure IVEzoom(x1: real, y1: real, x2: real, y2: real)

Arguments
- x1: The x coordinate of the lower left corner.
- y1: The y coordinate of the lower left corner.
- x2: The x coordinate of the upper right corner.
- y2: The y coordinate of the upper right corner.

Further Information
1. The viewable area is determined by its lower left and upper right corners.
2. This procedure only determines the automatic limits of the viewable area. The view and/or its scale can be changed by zooming or panning by using the mouse.
The Mosel ODBC interface provides a set of procedures and functions that may be used to access databases for which an ODBC driver is available. To use the ODBC interface, the following line must be included in the header of a Mosel model file:

```
uses 'mmodbc'
```

This manual describes the Mosel ODBC interface and shows how to use some standard SQL commands, but it is not meant to serve as a manual for SQL. The reader is referred to the documentation of the software he is using for more detailed information on these topics.

### 6.1 Example of use

Assume that the data source 'mydata' defines a database that contains a table 'pricelist' of the following form:

<table>
<thead>
<tr>
<th>articlenum</th>
<th>colour</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>blue</td>
<td>10.49</td>
</tr>
<tr>
<td>1002</td>
<td>red</td>
<td>10.49</td>
</tr>
<tr>
<td>1003</td>
<td>black</td>
<td>5.99</td>
</tr>
<tr>
<td>1004</td>
<td>blue</td>
<td>3.99</td>
</tr>
</tbody>
</table>

The following short example shows how to connect to a database from a Mosel model file, read in data, and disconnect from the data source:

```
model 'ODBC Example'
uses "mmodbc"

declarations
  prices: array(range) of real
end-declarations

setparam("SQLverbose",true)

SQLconnect("DSN=mydata")
writeln("Connection number: ",getparam("SQLconnection"))
SQLexecute("select articlenum,price from pricelist",prices)
```
Here the SQLverbose control parameter is set to true to enable ODBC message printing in case of error. Following the connection, the procedure SQLexecute is called to retrieve entries from the field price (indexed by field articlenum) in the table pricelist. Finally, the connection is closed.

### 6.2 Data transfer between Mosel and the Database

Data transfer between Mosel and the database is achieved by calls to the procedure SQLexecute. The value of the control parameter SQLndxcol and the type and structure of the second argument of the procedure decide how the data are transferred between the two systems.

#### From the Database to Mosel

Information is moved from the database to Mosel when performing a SELECT command for instance. Assuming mt has been declared as follows:

```mosel
mt: array(1..10,1..3) of integer
```

the execution of the call:

```mosel
SQLexecute("SELECT c1,c2,c3 from T",mt)
```

behaves differently depending on the value of SQLndxcol. If this control parameter is true, the columns c1 and c2 are used as indices and c3 is the value to be assigned. For each row (i,j,k) of the result set, the following assignment is performed by mmodbc:

```mosel
mt(i,j):=k
```

With a table T containing:

<table>
<thead>
<tr>
<th>c1</th>
<th>c2</th>
<th>c3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

we obtain the initialization:

```mosel
m2(1,2)=5, m(4,3)=6
```
If the control parameter *SQLndxcol* is false, all columns are treated as data. In this case, for each row \((i, j, k)\) the following assignments are performed:

\[
mt(r, 1) := i; \quad mt(r, 2) := j; \quad mt(r, 3) := k
\]

where \(r\) is the row number in the result set.

Here, the resulting initialization is:

- \(mt(1, 1) = 1, \quad mt(1, 2) = 2, \quad mt(1, 3) = 5\)
- \(mt(2, 1) = 4, \quad mt(2, 2) = 3, \quad mt(2, 3) = 6\)

The second argument of *SQLexecute* may also be an array of arrays. When using this version, the value of *SQLndxcol* is ignored and the first column(s) of the result set are always considered as indices and the following ones as values for the corresponding arrays. For instance, assuming we have the following declarations:

- \(m1, m2: \text{array}(1..10)\) of integer

with the statement:

\[
\text{SQLexecute}("\text{SELECT } c1, c2, c3 \text{ from T"}, [m1, m2])
\]

for each row \((i, j, k)\) of the result set, the following assignments are performed:

\[
m1(i) := j; \quad m2(i) := k
\]

So if we use the table \(T\) of our previous example, we get the initialization:

- \(m1(1) = 2, \quad m1(4) = 5\)
- \(m2(1) = 3, \quad m2(4) = 6\)

### From Mosel to the Database

Information is transferred from Mosel to the database when performing an INSERT command for instance. In this case, the way to use the Mosel arrays has to be specified by using parameters in the SQL command. These parameters are identified by the symbol '?' in the expression. For instance in the following expression 3 parameters are used:

\[
\text{INSERT INTO } T \text{ (c1,c2,c3) VALUES } (?,?,?)
\]
The command is then executed repeatedly as many times as the provided data allows to build new tuples of parameters. The initialization of parameters is similar to what is done for a SELECT statement.

Assuming mt has been declared as follows:

```plaintext
mt: array(1..2,1..3) of integer
```

and initialized with this assignment:

```plaintext
mt := [1,2,3,
      4,5,6]
```

the execution of the call:

```plaintext
SQLexecute("INSERT INTO T (c1,c2,c3) VALUES (?,?,?)",mt)
```

behaves differently depending on the value of SQLndxcol. If this control parameter is true, for each execution of the command, the following assignments are performed by mmodbc (?1,?2,?3 denote respectively the first second and third parameter):

```plaintext
'?1':= i, '?2':= j, '?3':= mt(i,j)
```

The execution is repeated for all possible values of i and j (in our example 6 times). The resulting table T is therefore:

```plaintext
c1  c2  c3
1   1   1
1   2   2
1   3   3
2   1   4
2   2   5
2   3   6
```

If the control parameter SQLndxcol is false, only the values of the Mosel array are used to initialize the parameters. So, for each execution of the command, we have:

```plaintext
'?1':=mt(i,1), '?2':=mt(i,2), '?3':=mt(i,3)
```

The execution is repeated for all possible values of i (in our example 2 times). The resulting table T is therefore:
When `SQLexecute` is used with an array of arrays, the behavior is again similar to that described earlier for the SELECT command: the first parameter(s) are assigned index values and the final ones the actual array array values. For instance, assuming we have the following declarations:

```
m1, m2: array(1..3) of integer
```

and the arrays have been initialized as follows:

```
m1 := [1, 2, 3]
m2 := [4, 5, 6]
```

then the following call:

```
SQLexecute("INSERT INTO T (c1,c2,c3) VALUES (?,?,?)", [m1, m2])
```

executes the INSERT command 3 times. For each execution, the following parameter assignments are performed:

```
'?1': = i, '?2' := m1(i), '?3' := m2(i)
```

The resulting table `T` is therefore:

```
c1  c2  c3  
1    1    4  
2    2    5  
3    3    6  
```

### 6.3 ODBC and MS Excel

Microsoft Excel is a spreadsheet application. Since ODBC was primarily designed for databases special rules have to be followed to read and write Excel data using ODBC:

- A table of data is referred as either a named range (e.g. `MyRange`), a worksheet name (e.g. `[Sheet1$]`) or an explicit range (e.g. `[Sheet1$B2:C12]`)
- By default, the first row of a range is used for naming the columns (to be used in SQL statements). The option `FIRSTROWHASNAMES=0` disables this feature and
columns are implicitly named F1, F2... However, even with this option, the first row is ignored and cannot contain data.

- The data types of columns are deduced by the Excel driver by scanning the first 8 rows. The number of rows analysed can be changed using the option `MAXSCANROWS=n` (n between 1 and 8).

It is important to be aware that when writing to database tables specified by a named range in Excel, they will increase in size if new data are added using an INSERT statement. To overwrite existing data in the worksheet, the SQL statement UPDATE can be used in most cases (although this command is not fully supported). Now suppose that we wish to write further data over the top of data that have already been written to a range using an INSERT statement. Within Excel it is not sufficient to delete the previous data by selecting them and hitting the Delete key. If this is done, further data will be added after a blank rectangle where the deleted data used to reside. Instead, it is important to use Edit/Delete/Shift cells up within Excel, which will eliminate all traces of the previous data, and the enlarged range.

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

When first experimenting with acquiring or writing data via ODBC it is tempting to use short names for column headings. This can lead to horrible-to-diagnose errors if you inadvertently use an SQL keyword. We strongly recommend that you use names like ‘myParameters’, or ‘myParams’, or ‘myTime’, which will not clash with SQL reserved keywords.
6.4 Control Parameters

**SQLbufsize**

**Description:** Size in kilobytes of the buffer used for exchanging data between Mosel and the ODBC driver.

**Type:** Integer, read/write

**Values:** At least 1 (default: 4)

**Affects Routines:** SQLexecute, SQLreadstring.

**SQLcolsise**

**Description:** Maximum length of strings accepted to exchange data. Anything exceeding this size is cut off.

**Type:** Integer, read/write

**Values:** 8 to 1024 (default: 64)

**Affects Routines:** SQLexecute, SQLreadstring.

**SQLconnection**

**Description:** Identification number of the active ODBC connection. By changing the value of this, it is possible to work with several connections simultaneously. Set by SQLconnect.

**Type:** Integer, read/write

**Affects Routines:** SQLdisconnect, SQLexecute, SQLreadinteger, SQLreadreal, SQLreadstring.
**SQLndxcol**

**Description**
Indicates whether the first columns of each row must be interpreted as indices in all cases. Setting it to the value `false` might be useful if, for example, one is trying to access a non-relational table, perhaps a dense spreadsheet table.

**Type**
Boolean, read/write

**Values**
- `true` interpret the first columns of each row as indices (default);
- `false` do not interpret the first columns of each row as indices.

**Affects Routines**
SQLexecute, SQLreadinteger, SQLreadreal, SQLreadstring.

---

**SQLrowcnt**

**Description**
Number of lines affected by the last SQL command.

**Type**
Integer, read only

**Set by Routines**
SQLexecute, SQLreadinteger, SQLreadreal, SQLreadstring.

---

**SQLrowxfr**

**Description**
Number of lines transferred during the last SQL command.

**Type**
Integer, read only

**Set by Routines**
SQLexecute, SQLreadinteger, SQLreadreal, SQLreadstring.

---

**SQLsuccess**

**Description**
Indicates whether the last SQL command has been executed successfully.

**Type**
Boolean, read only
### SQLverbose

**Description**
Enables or disables message printing by the ODBC driver.

**Type**
Boolean, read/write

**Values**
- true: enable message printing;
- false: disable message printing (default).

### 6.5 Procedures and Functions

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Pg</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLconnect</td>
<td>Connect to a data source.</td>
<td>118</td>
</tr>
<tr>
<td>SQLdisconnect</td>
<td>Disconnect from a data source.</td>
<td>119</td>
</tr>
<tr>
<td>SQLexecute</td>
<td>Execute an SQL query on the data source.</td>
<td>120</td>
</tr>
<tr>
<td>SQLreadinteger</td>
<td>Read an integer value from the data source.</td>
<td>122</td>
</tr>
<tr>
<td>SQLreadreal</td>
<td>Reads a real value from the data source.</td>
<td>123</td>
</tr>
<tr>
<td>SQLreadstring</td>
<td>Reads a string value from the data source.</td>
<td>124</td>
</tr>
<tr>
<td>SQLupdate</td>
<td>Update the selected data with the provided array(s).</td>
<td>125</td>
</tr>
</tbody>
</table>
**SQLconnect**

**Purpose**
Establish a connection to a data source.

**Synopsis**

procedure SQLconnect(s: string)

**Arguments**

- **s**  
  Connection string.

**Example**

The following connects to the ‘MySQL’ database ‘test’ as the user ‘yves’ with password ‘DaSH’:

```plaintext
SQLconnect("DSN=mysql;DB=test;UID=yves;PWD=DaSH")
```

**Further Information**

It is possible to open several connections, but the connection opened last becomes active. Each connection is assigned an identification number, which can be obtained by getting the value of the control parameter `SQLconnection` after this procedure has been executed. This parameter can also be used to change the active connection.

**Related Topics**

SQLdisconnect.
**SQLdisconnect**

**Purpose**
Terminate the active database connection

**Synopsis**
```
procedure SQLdisconnect
```

**Further Information**
The active connection can be changed by setting the control parameter `SQLconnection`.

**Related Topics**
`SQLconnect`.
SQLexecute

Purpose
Execute an SQL command.

Synopsis
procedure SQLexecute(s: string)
procedure SQLexecute(s: string, a: array)
procedure SQLexecute(s: string, m: set)

Arguments
s SQL command to be executed.
a An array of one of the basic types (integer, real, string or boolean). May be a tuple of arrays.
m A set of one of the basic types (integer, real, string or boolean).

Example
The following examples contains four SQLexecute statements performing the following tasks:
- get all different values of the column colour in the table pricelist;
- initialize the arrays colours and prices with the values of the columns colour and price of the table pricelist;
- create a new table newtab in the active database with two columns, ndx and price;
- add data entries to the table newtab.
  declarations
  prices: array(1001..1004) of real
  colours: array(1001..1004) of string
  allcolours: set of string
  end-declarations
  ...
  SQLexecute("select colour from pricelist",allcolours)
  SQLexecute(
    "select articlenum,colour,price from pricelist",
    [colours,prices])
  SQLexecute(
    "create table newtab (ndx integer, price double")
  SQLexecute(}
"insert into table newtab (ndx, price) value (?,?), prices"

Further Information
The user is referred to the documentation for the database driver being used for more information about the commands that it supports.

Related Topics
SQLupdate, SQLreadinteger, SQLreadreal, SQLreadstring.
**SQLreadinteger**

**Purpose**
Read an integer value from the active database.

**Synopsis**
```pascal
function SQLreadinteger(s: string): integer
```

**Arguments**
- `s` SQL command for selecting the value to be read.

**Return Value**
Integer value read or 0.

**Example**
The following gets article number of the first data item in table pricelist for which the field colour is set to blue:
```pascal
i:= SQLreadinteger(
  "select articlenum from pricelist where colour=blue"
)
```

**Further Information**
1. 0 is returned if no integer value can be found.
2. If the given SQL selection command does not denote a single value, the first value to which the selection criterion applies is returned.

**Related Topics**
SQLexecute, SQLreadreal, SQLreadstring.
SQLreadreal

Purpose
Read a real value from the active database.

Synopsis
function SQLreadreal(s: string): real

Arguments
s    SQL command for selecting the value to be read.

Return Value
Real value read or 0.

Example
The following returns the price of the data item with index 2 in the table newtab:
   r := SQLreadreal("select price from newtab where ndx=2")

Further Information
1. 0 is returned if no real value can be found.
2. If the given SQL selection command does not denote a single value, the first value to which the selection criterion applies is returned.

Related Topics
SQLexecute, SQLreadinteger, SQLreadstring.
**SQLreadstring**

**Purpose**
Read a string from the active database.

**Synopsis**
function SQLreadstring(s: string): string

**Arguments**
- `s` SQL command for selecting the string to be read.

**Return Value**
String read or empty string.

**Example**
The following retrieves the colour of the (first) data item in table `pricelist` with article number 1004:

```mosel
s := SQLreadstring(
  "select colour from pricelist where articlenum=1004"
)
```

**Further Information**
1. The empty string is returned if no string value can be found.
2. If the given SQL selection command does not denote a single entry, the first string to which the selection criterion applies is returned.

**Related Topics**
SQLexecute, SQLreadinteger, SQLreadreal.
**SQLUpdate**

**Purpose**
Update the selected data with the provided array(s).

**Synopsis**

```
procedure SQLupdate(s: string, a: array)
```

**Arguments**
- `s` An SQL ‘SELECT’ command.
- `a` An array of one of the basic types (integer, real, string or boolean). May be a tuple of arrays.

**Example**
The following example initializes the array prices with the values of the table pricelist, change some values in the array and finally, updates the data in the table pricelist:

```
declarations
  prices: array(1001..1004) of real
end-declarations
SQLexecute("select articlenum,price from pricelist",
            prices)
prices(1002):=prices(1002)*0.9
prices(1003):=prices(1003)*0.8
SQLupdate("select articlenum,price from pricelist",
           prices)
```

**Further Information**
This procedure updates the data selected by an SQL command (usually ‘SELECT’) with an array or tuple of arrays. This procedure is available only if the data source supports positioned updates (for instance, MS Access does but MS Excel does not).

**Related Topics**
- SQLexecute.
7 mmquad

The module mmquad extends the Mosel language with a new type for representing quadratic expressions. To use this module, the following line must be included in the header of the Mosel model file:

```mosel
uses 'mmquad'
```

The first part of this chapter presents the new functionality for the Mosel language that is provided by mmquad, namely the new type qexp and a set of subroutines that may be applied to objects of this type.

Via the inter-module communication interface, the module mmquad publishes several of its library functions. These are documented in the second part. By means of an example it is shown how the functions published by mmquad can be used in another module to access quadratic expressions and work with them.

7.1 New Functionality for the Mosel Language

The Type qexp and Its Operators

The module mmquad defines the type qexp to represent quadratic expressions in the Mosel Language. As shown in the following example, mmquad also defines the standard arithmetic operations that are required for working with objects of this type. By and large, these are the same operations as for linear expressions (type linctr of the Mosel language) with in addition the possibility to multiply two decision variables or one variable with itself. For the latter, the exponential notation $x^2$ may be used (assuming that $x$ is of type mpvar).

Example: Using mmquad for Quadratic Programming

Quadratic expressions as defined with the help of mmquad may be used to define quadratic objective functions for Quadratic Programming (QP) or Mixed Integer Quadratic Programming (MIQP) problems. The Xpress-Optimizer module mmxprs for instance accepts expressions of type qexp as arguments for its optimization subroutines minimize and maximize, and for the procedure loadprob (see also the mmxprs Reference Manual).

```mosel
model "Small MIQP example"
uses "mmxprs", "mmquad"
```
declarations
  x: array(1..4) of mpvar
Obj: qexp
end-declarations

! Define some linear constraints
x(1) + 2*x(2) - 4*x(4) >= 0
3*x(1) - 2*x(3) - x(4) <= 100
x(1) + 3*x(2) + 3*x(3) - 2*x(4) => 10
2  <=  x(1);  x(1)  <=  20
x(2) is_integer; x(3) is_integer
x(4) is_free

! The objective function is a quadratic expression
Obj:= x(1) + x(1)^2 + 2*x(1)*x(2) + 2*x(2)^2 + x(4)^2

! Solve the problem and print its solution
minimize(Obj)
writeln("Solution: ", getobjval)
forall(i in 1..4) writeln(getsol(x(i)))
end-model

7.2 Procedures and Functions

The module mmquad overloads certain subroutines of the Mosel language, replacing an argument of type linctr by the type qexp.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>Pg</th>
</tr>
</thead>
<tbody>
<tr>
<td>exportprob</td>
<td>Export a quadratic problem to a file</td>
<td>129</td>
</tr>
<tr>
<td>getsol</td>
<td>Get the solution value of a quadratic expression</td>
<td>130</td>
</tr>
</tbody>
</table>
exportprob

**Purpose**
Export a quadratic problem to a file.

**Synopsis**
```
procedure exportprob(options integer, filename: string, obj: qexp)
```

**Arguments**
- `options` File format options:
  - `EP_MIN` LP format, minimization (default);
  - `EP_MAX` LP format, maximization;
  - `EP_MPS` MPS format;
  - `EP_STRIP` use scrambled names.
  Several options may be combined using `+`.
- `filename` Name of the output file. If the empty string "" is given, output is printed to the standard output (the screen).
- `obj` Objective function (quadratic expression)

**Example**
The following example prints the problem to screen using the default format, and then exports the problem in LP-format to file 'probl.lp' maximizing constraint Profit:
```
uses "mmquad"
declarations
  Profit:qexp
end-declarations
...
exportprob(0, "", Profit)
exportprob(EP_MAX, "probl", Profit)
```

**Further Information**
This procedure overloads the exportprob subroutine of Mosel to handle quadratic objective functions. It exports the current problem to a file, or if no file name is given (empty string ""), prints it on screen. If the given filename has no extension, mmquad appends .lp to it for LP format files and .mat for MPS format.
**getsol**

**Purpose**
Get the solution value of a quadratic expression.

**Synopsis**

```plaintext
function getsol(q: qexp):real
```

**Arguments**

- `q` A quadratic expression

**Return Value**
Solution value or 0.

**Further Information**
This function returns the evaluation of a given quadratic expression using the current (primal) solution values of its variables. Note that the solution value of a variable is 0 if the problem has not been solved or the variable is not contained in the problem that has been solved.
7.3 Published Library Functions

The module mmquad publishes some of its library functions via the service IMCI for use by other modules (see the Mosel Native Interface Reference Manual for more detail about services). The list of published functions is contained in the interface structure mmquad_imci that is defined in the module header file mmquad.h.

From another module, the context of mmquad and its communication interface can be obtained using functions of the Mosel Native Interface as shown in the following example.

```c
static XPRMnifct mm;
XPRMcontext mmctx;
XPRMdsolib dao;
mmquad_imci mq;
void **quadctx;

dso=mm->finddso("mmquad"); /* Retrieve the mmquad module*/
/* Get the module context and the communication interface of mmquad */
quadctx=*(mm->getdsoctx(mmctx, dso, (void **)(&mq)));
```

Typically, a module calling functions that are provided by mmquad will include this module into its list of dependencies in order to make sure that mmquad will be loaded by Mosel at the same time as the calling module. The 'dependency' service of the Mosel Native Interface has to be used to set the list of module dependencies:

```c
/* Module dependency list */
static const char *deplist[]={"mmquad",NULL};
static XPRMdsoserv tabserv[]= /* Table of services */
{
  {XPRM_SRV_DEPLST, (void *)deplist}
};
```

Complete Module Example

If the Mosel procedures write/writeln are applied to a quadratic expression, they print the address of the expression and not its contents (just the same would happen for types mpvar or linctr). Especially for debugging purposes, it may be useful to be able to display some more detailed information. The module example printed below defines the procedure printqexp that displays all the terms of a quadratic
expression (for simplicity’s sake, we do not retrieve the model names for the variables but simply print their addresses).

model "Test printqexp module"
uses "printqexp"
declarations
  x: array(1..5) of mpvar
  q: qexp
end-declarations

  printqexp(10+x(1)*x(2)-3*x(3)^2)
  q:= x(1)*(sum(i in 1..5) i*x(i))
  printqexp(q)
end-model

Note that in this model it is not necessary to load explicitly the mmquad module. This will be done by the printqexp module because mmquad appears in its dependency list.

#include <stdlib.h>
#include "xprm_ni.h"
#include "mmquad.h"

/*** Function prototypes ***/
static int printqexp(XPRMcontext ctx,void *libctx);

/*** Structures for passing info to Mosel ***/
/* Subroutines */
static XPRMdsofct tabfct[] =
  {"printqexp", 1000, XPRM_TYP_NOT, 1, "|qexp|", printqexp}
};

/* Module dependency list */
static const char *deplist[]="mmquad",NULL);
/* Services */
static XPRMdsoserv tabserv[] =
  {XPRM_SRV_DEPLST, (void *)deplist}
};
/* Interface structure */
static XPRMdsointer dsointer =
  {

0, NULL, sizeof(tabfct)/sizeof(XPRMdsofct), tabfct,
0, NULL, sizeof(tabserv)/sizeof(XPRMdsobserv), tabserv
};

/**** Structures used by this module ****/
static XPRMnifct mm; /* For storing Mosel NI function table */

/**** Initialize the module library just after loading it ****/
DSO_INIT printqexp_init(XPRMnifct nifct, int *interver, int *libver,
XPRMdsointer **interf)
{
    mm = nifct; /* Save the table of Mosel NI functions */
    *interver = MM_NIVERS; /* Mosel NI version */
    *libver = MM_MKVER(0, 0, 1); /* Module version */
    *interf = &dsointer; /* Pass info about module contents to Mosel */
    return 0;
}

/**** Implementation of "printqexp" ****/
static int printqexp(XPRMcontext ctx, void *libctx)
{
    XPRMdsolib dso;
    mmquad_imci mq;
    mmquad_qexp q;
    void **quadctx;
    void *prev;
    XPRMmpvar v1, v2;
    double coeff;
    int nlin, i;

    /* Retrieve reference to the mmquad module*/
    dso = mm->finddso("mmquad");
    /* Get the module context and the communication interface of mmquad */
    quadctx = *(mm->getdsoctx(ctx, dso, (void **)(&mq)));
    /* Get the quadratic expression from the stack */
    q = XPRM_POP_REF(ctx);
    /* Get the number of linear terms */
    mq->getqexpstat(ctx, quadctx, q, &nlin, NULL, NULL, NULL);
    /* Get the first term (constant) */
    prev = mq->getqexpnextterm(ctx, quadctx, q, NULL, &v1, &v2, &coeff);
    if (coeff != 0) mm->printf(ctx, "%g %p", coeff);
    for (i = 0; i < nlin; i++) /* Print all linear terms */
    {
        prev = mq->getqexpnextterm(ctx, quadctx, q, prev, &v1, &v2, &coeff);
        mm->printf(ctx, "%g %p", coeff, v2);
while(prev!=NULL) /* Print all quadratic terms */
{
    prev=mq->getqexpnextterm(ctx, quadctx, q, prev, &v1, &v2, &coeff);
    mm->printf(ctx,"%+g %p * %p ", coeff, v1, v2);
}
mm->printf(ctx,"\n");
return XPRM_RT_OK;

Description of the Library Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Pg</th>
</tr>
</thead>
<tbody>
<tr>
<td>getqexpstat</td>
<td>Get information about a quadratic expression.</td>
<td>135</td>
</tr>
<tr>
<td>clearqexpstat</td>
<td>Free the memory allocated by getqexpstat.</td>
<td>136</td>
</tr>
<tr>
<td>getqexpnextterm</td>
<td>Enumerate the list of terms contained in a quadratic expression.</td>
<td>137</td>
</tr>
</tbody>
</table>
getqexpstat

**Purpose**
Get information about a quadratic expression.

**Synopsis**
```c
int getqexpstat(XPRMctx ctx, void *quadctx, mmquad_qexp q,
    int *nblin, int *nbqd, int *changed, XPRMmpvar **lsvar);
```

**Arguments**
- **ctx** Mosel's execution context
- **quadctx** Context of mmquad
- **q** Reference to a quadratic expression
- **nblin** Pointer to which the number of linear terms is returned (may be `NULL`)
- **nbqd** Pointer to which the number of quadratic terms is returned (may be `NULL`)
- **changed** Pointer to which the number of quadratic terms is returned (may be `NULL`). Possible values:
  - 1 : the expression `q` has been modified since the last call to this function
  - 0 otherwise
- **lsvar** Pointer to which is returned the table of variables that appear in the quadratic expression `q` (may be `NULL`)

**Return Value**
Total number of terms in the expression

**Further Information**
This function returns in its arguments information about a given quadratic expression. Any of these arguments may be `NULL` to indicate that the corresponding information is not required. The last entry of the table `lsvar` is `NULL` to indicate its end. This table is allocated by the module mmquad, it must be freed by the next call to this function or with function `clearqexpstat`. 
clearqexpstat

Purpose
Free the memory allocated by getqexpstat.

Synopsis
void clearqexpstat(XPRMctx ctx, void *quadctx);

Arguments
ctx Mosel's execution context
quadctx Context of mmquad
getqexpnextterm

Purpose
Enumerate the list of terms contained in a quadratic expression.

Synopsis
void *getqexpnextterm(XPRMctx ctx, void *quadctx,
    mmquad_qexp q, void *prev, XPRMmpvar *v1, XPRMmpvar *v2,
    double *coeff);

Arguments
ctx Mosel’s execution context
quadctx Context of mmquad
q Reference to a quadratic expression
prev Last value returned by this function. Should be NULL for the first call
v1,v2 Pointers to return the decision variable references for the current term
coeff Pointer to return the coefficient of the current term

Return Value
The value to be used as prev for the next call or NULL when all terms have been returned

Further Information
This function can be called repeatedly to enumerate all terms of a quadratic expression. For the first call, the parameter prev must be NULL and the function returns the constant term of the quadratic expression (for v1 and v2 the value NULL is returned and coeff contains the constant term). For the following calls, the value of prev must be the last value returned by the function. The enumeration is completed when the function returns NULL. If this function is called repeatedly, after the constant term it returns next all linear terms and then the quadratic terms.
The mmsystem module provides a set of procedures and functions related to the operating system. Due to the nature of these commands, it should be clear that behaviour may vary between systems and some care should be exercised during use when writing portable code. To use this module, the following line must be included in the header of the Mosel model file:

\texttt{uses 'mmsystem'}

### 8.1 Procedures and Functions

In general, the procedures and functions of mmsystem do not fail but set a status variable that can be read with \texttt{getsysstat}. To make sure the operation has been performed correctly, check the value of this variable after each system call.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>Pg</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdelete</td>
<td>Delete a file.</td>
<td>140</td>
</tr>
<tr>
<td>fmove</td>
<td>Move a file.</td>
<td>141</td>
</tr>
<tr>
<td>getcwd</td>
<td>Get the current working directory.</td>
<td>142</td>
</tr>
<tr>
<td>getenv</td>
<td>Get the value of an environment variable.</td>
<td>143</td>
</tr>
<tr>
<td>getfstat</td>
<td>Get the status of a file or directory.</td>
<td>144</td>
</tr>
<tr>
<td>getsysstat</td>
<td>Get the system status.</td>
<td>145</td>
</tr>
<tr>
<td>gettime</td>
<td>Get a time measure in seconds.</td>
<td>146</td>
</tr>
<tr>
<td>makedir</td>
<td>Create a directory.</td>
<td>147</td>
</tr>
<tr>
<td>removedir</td>
<td>Remove a directory.</td>
<td>148</td>
</tr>
<tr>
<td>system</td>
<td>Execute a command on the operating system.</td>
<td>149</td>
</tr>
</tbody>
</table>
**fdelete**

**Purpose**
Delete a file.

**Synopsis**
procedure fdelete(filename: string)

**Arguments**
filename  The name and path of the file to be deleted.

**Related Topics**
removedir.
**fmove**

**Purpose**
Rename or move a file.

**Synopsis**
```plaintext
procedure fmove(namesrc: string, namedest: string)
```

**Arguments**
- `namesrc` The name and path of the file to be moved or renamed.
- `namedest` The destination name and/or path.

**Further Information**
This procedure renames the file `namesrc` to `namedest`. If the second name is a directory, the file is moved into that directory.
getcwd

Purpose
Get the current working directory.

Synopsis
function getcwd: string

Return Value
Current working directory.

Further Information
This function returns the current working directory, that is the directory where Mosel is being executed and where files are looked for.
**getenv**

**Purpose**
Get the value of an environment variable of the operating system.

**Synopsis**
function getenv(name: string): string

**Arguments**

- name Name of the environment variable.

**Return Value**
Value of the environment variable (an empty string if the variable is not defined).

**Example**
The following returns the value of the PATH environment variable:
```mosel
str := getenv("PATH")
```
getfstat

Purpose
Get the status (type and access mode) of a file or directory.

Synopsis
function getfstat(filename: string): integer

Arguments
filename Name (and path) of the file or directory to check.

Return Value
Bit encoded file status.

Example
The following determines whether ftest is a directory and if it is writable:

```pascal
fstat := getfstat("ftest")
if bittest(fstat,SYS_TYP)=SYS_DIR then
  writeln("ftest is a directory")
end-if

if bittest(fstat,SYS_WRITE)=SYS_WRITE then
  writeln("ftest is writeable")
end-if
```

Further Information
The status type returned may be decoded using the constant mask SYS_TYP (the types are exclusive). Possible values are:

- SYS_DIR directory;
- SYS_REG regular file;
- SYS_OTH special file (device, pipe...).

The access mode may be decoded using the constant mask SYS_MOD (the access modes are additive). Possible values are:

- SYS_READ can be read;
- SYS_WRITE can be modified;
- SYS_EXEC is executable.
getsysstat

Purpose
Get the system status.

Synopsis
function getsysstat: integer

Return Value
0 if the last operation of the module was executed successfully.

Example
In this example, we attempt to delete the file randomfile. If this is unsuccessful, a warning message is displayed:

fdelete("randomfile")
if getsysstat <> 0 then
  writeln("randomfile could not be deleted.")
end-if
**gettime**

**Purpose**
Get a time measure in seconds.

**Synopsis**
function gettime: real

**Return Value**
Time measure in seconds.

**Example**
The following prints the program execution time:
```
starttime := gettime
...
write("Time: ", gettime-starttime)
```

**Further Information**
The absolute value is system-dependent, which means that the elapsed time of a program has to be computed relative to its start time.
makedir

Purpose
Create a new directory in the file system.

Synopsis
procedure makedir(dirname: string)

Arguments
dirname The name and path of the directory to be created.

Related Topics
removedir.
removedir

Purpose
Remove a directory.

Synopsis
procedure removedir(dirname: string)

Arguments
dirname The name and path of the directory to delete.

Further Information
For deletion of a directory to succeed, the given directory must be empty.

Related Topics
fdelete, makedir.
system

Purpose
Execute a system command.

Synopsis
procedure system(command: string)

Arguments
command  The command to be executed.

Example
The following creates the directory Newdir:
system("mkdir Newdir")

Further Information
This procedure should be avoided in applications that are to be run on
different systems because such a call is always system dependent and may not
be portable.
9 mmxprs

The mmxprs module provides access to the Xpress-Optimizer from within a Mosel model and as such it requires the Xpress-Optimizer libraries to be installed on the system. To use this module, the following line must be included in the header of the Mosel model file:

```plaintext
uses 'mmxprs'
```

A large number of optimization-related routines are provided, ranging from those for finding a solution to the problem, to those for setting callbacks and cut manager functions. Whilst a description of their usage is provided in this chapter, further details relating to the usage of these may be found by consulting the Xpress-Optimizer Reference Manual.

9.1 Control Parameters

This module also extends the getparam function and the setparam procedures in order to access all the controls and problem attributes of the Optimizer (for example the problem attribute LPSTATUS is mapped to the mmxprs control parameter XPRS_lpstatus). In addition to these, the following control parameters are also defined:

<table>
<thead>
<tr>
<th>XPRS_colorder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
</tr>
<tr>
<td>Type</td>
</tr>
<tr>
<td>Values</td>
</tr>
</tbody>
</table>
**XPRS_loadnames**

**Description**
Enables or disables the loading of MPS names into the Optimizer.

**Type**
Boolean, read/write

**Values**
- true: enables loading of names;
- false: disables loading of names (default).

**Affects Routines**
loadprob, maximize, minimize.

**XPRS_problem**

**Description**
The Optimizer problem pointer. This attribute is only required in applications using both Mosel and the Optimizer at the C level.

**Type**
Integer, read only

**XPRS_probname**

**Description**
Read/set the problem name used by the Optimizer to build its working files (this name may contain a full path). If set to the empty string (default value), a unique name with a path to the temporary directory of the OS is generated.

**Type**
string, read/write

**XPRS_verbose**

**Description**
Enables or disables message printing by the Optimizer.

**Type**
Boolean, read/write

**Values**
- true: enable message printing;
- false: disable message printing (default).
## 9.2 Procedures and Functions

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>Pg</th>
</tr>
</thead>
<tbody>
<tr>
<td>clearmipdir</td>
<td>Delete all defined MIP directives.</td>
<td>154</td>
</tr>
<tr>
<td>clearmodcut</td>
<td>Delete all defined model cuts.</td>
<td>155</td>
</tr>
<tr>
<td>delbasis</td>
<td>Delete a previously-saved basis.</td>
<td>156</td>
</tr>
<tr>
<td>getiis</td>
<td>Compute then get IIS.</td>
<td>157</td>
</tr>
<tr>
<td>getlb</td>
<td>Get the lower bound for a variable.</td>
<td>158</td>
</tr>
<tr>
<td>getprobstat</td>
<td>Get the Optimizer problem status.</td>
<td>159</td>
</tr>
<tr>
<td>getub</td>
<td>Get the upper bound for a variable.</td>
<td>160</td>
</tr>
<tr>
<td>initglobal</td>
<td>Initialise the global search.</td>
<td>161</td>
</tr>
<tr>
<td>loadbasis</td>
<td>Load a previously-saved basis.</td>
<td>162</td>
</tr>
<tr>
<td>loadprob</td>
<td>Load a problem into the Optimizer.</td>
<td>163</td>
</tr>
<tr>
<td>maximize</td>
<td>Maximize the objective function.</td>
<td>164</td>
</tr>
<tr>
<td>minimize</td>
<td>Minimize the objective function.</td>
<td>164</td>
</tr>
<tr>
<td>readbasis</td>
<td>Read basis from a file.</td>
<td>166</td>
</tr>
<tr>
<td>readdirs</td>
<td>Read directives from a file.</td>
<td>167</td>
</tr>
<tr>
<td>savebasis</td>
<td>Save a basis.</td>
<td>168</td>
</tr>
<tr>
<td>setcallback</td>
<td>Set up a callback function.</td>
<td>169</td>
</tr>
<tr>
<td>setlb</td>
<td>Set the lower bound for a variable.</td>
<td>171</td>
</tr>
<tr>
<td>setmipdir</td>
<td>Set a directive.</td>
<td>172</td>
</tr>
<tr>
<td>setmodcut</td>
<td>Mark a constraint as a model cut.</td>
<td>173</td>
</tr>
<tr>
<td>setub</td>
<td>Set the upper bound for a variable.</td>
<td>174</td>
</tr>
<tr>
<td>writebasis</td>
<td>Write the current basis to a file.</td>
<td>175</td>
</tr>
<tr>
<td>writedirs</td>
<td>Write the current directives to a file.</td>
<td>176</td>
</tr>
<tr>
<td>writeprob</td>
<td>Write the current problem in the optimizer to a file.</td>
<td>177</td>
</tr>
</tbody>
</table>
clearmipdir

Purpose
Delete all defined MIP directives.

Synopsis
procedure clearmipdir

Related Topics
setmipdir
clearmodcut

**Purpose**
Delete all defined model cuts.

**Synopsis**
```plaintext
procedure clearmodcut
```

**Related Topics**
- `setmodcut`. 
delbasis

Purpose
Delete a previously saved basis.

Synopsis
procedure delbasis(num: integer)
procedure delbasis(name: string)

Arguments
num Reference number of a previously saved basis.
name Name of a previously saved basis.

Example
The following saves a basis, giving it the reference number 2, performs other tasks and finally deletes the basis with reference number 2:

savebasis(2)
...
delbasis(2)

Related Topics
loadbasis, savebasis.
**getiis**

**Purpose**
Compute then get the Irreductible Infeasible Sets (IIS).

**Synopsis**
```verbatim
procedure getiis(vset: set of mpvar, cset: set of linctr)
```

**Arguments**
- `vset`  Set to return the decision variables of the IIS.
- `cset`  Set to return the constraints of the IIS.

**Further Information**
This procedure computes the IIS then stores the result in the provided parameters. The sets passed to this procedure are not reset before being used.
getlb

Purpose
Get the lower bound of a variable.

Synopsis
function getlb(x: mpvar): real

Arguments
x A decision variable.

Return Value
Lower bound of the variable.

Further Information
This function returns the lower bound of a variable that is currently held by the Optimizer. The bound value may be changed directly in the Optimizer using setlb. Changes to the variable in Mosel are not taken into account by this function unless the problem has been reloaded since (procedure loadprob).

Related Topics
getub, setlb, setub.
getprobstat

Purpose
Get the Optimizer problem status.

Synopsis
function getprobstat: integer

Return Value
Status of the problem currently held in the Optimizer:
XPRS_OPT solved to optimality;
XPRS_UNF unfinished;
XPRS_INF infeasible;
XPRS_UNB unbounded.

Example
The following procedure displays the current problem status:
procedure print_status
declarations
status:array({XPRS_OPT, XPRS_UNF, XPRS_INF, XPRS_UNB})
of string
end-declarations

status:=['Optimum found', 'Unfinished', 'Infeasible', 'Unbounded']
writeln(status(getprobstat))
end-procedure

Further Information
More detailed information than that provided by this function can be obtained with function getparam, retrieving the problem attributes XPRS_presolvestate, XPRS_lpstatus and XPRS_mipstatus (see the Xpress-Optimizer Reference Manual).
getub

Purpose
Get the upper bound of a variable.

Synopsis
function getub(x: mpvar): real

Arguments
x A decision variable.

Return Value
Upper bound of the variable.

Further Information
The bound value may be changed directly in the Optimizer using setub. Changes to the variable in Mosel are not taken into account by this function unless the problem has been reloaded since (procedure loadprob).

Related Topics
getlb, setlb, setub,
**initglobal**

**Purpose**
Reset the global search started by maximize or minimize.

**Synopsis**

```plaintext
procedure initglobal
```

**Related Topics**

maximize, minimize
**loadbasis**

**Purpose**
Load a basis into the Optimizer that has previously been saved using the procedure `savebasis`.

**Synopsis**
```
procedure loadbasis(num: integer)
procedure loadbasis(name: string)
```

**Arguments**
- **num**: Reference number of a previously saved basis.
- **name**: Name of a previously saved basis.

**Example**
The following saves a basis, giving it reference number 2, changes the problem and then loads it into the Optimizer, reloading the old basis:
```
declarations
    Mincost: linctr
end-declarations

savebasis(2)
...
loadprob(MinCost)
loadbasis(2)
```

**Further Information**
1. The problem must be loaded in the Optimizer for `loadbasis` to have any effect. If this has not recently been carried out using `maximize` or `minimize`, it must be explicitly loaded using `loadprob`.
2. Unless the basis is deleted (procedure `delbasis`), it may be loaded repeatedly with this procedure.

**Related Topics**
delbasis, savebasis.
loadprob

Purpose
Load a problem into the optimizer.

Synopsis
procedure loadprob(obj: linctr)
procedure loadprob(force: boolean, obj: linctr)
procedure loadprob(obj: linctr, extravar: set of mpvar)
procedure loadprob(force: boolean, obj: linctr,
                     extravar: set of mpvar)
procedure loadprob(qobj: qexp)
procedure loadprob(qobj: qexp, extravar: set of mpvar)

Arguments
- obj: Objective function constraint.
- qobj: Quadratic objective function (with module mmquad).
- force: Load the matrix even if not required.
- extravar: Extra variables to include.

Further Information
1. This procedure explicitly loads a problem into the Optimizer. It is called automatically by the optimization procedures minimize and maximize if the problem has been modified in Mosel since the last call to the Optimizer. Nevertheless in some cases, namely before loading a basis, it may be necessary to reload the problem explicitly using this procedure. If the problem has not been modified since the last call to loadprob, the problem is not reloaded into the Optimizer. The argument 'force' can be used to force a reload of the problem in such a case. The argument 'extravar' is a set of variables to be included into the problem even if they do not appear in any constraint (i.e. they become empty columns in the matrix).
2. Support for quadratic programming requires the module mmquad

Related Topics
maximize, minimize.
maximize, minimize

**Purpose**
Maximize (minimize) the current problem.

**Synopsis**
- `procedure maximize(alg: integer, obj: linctr)`
- `procedure maximize(obj: linctr)`
- `procedure maximize(alg: integer, qobj: qexp)`
- `procedure maximize(qobj: qexp)`
- `procedure minimize(alg: integer, obj: linctr)`
- `procedure minimize(obj: linctr)`
- `procedure minimize(alg: integer, qobj: qexp)`
- `procedure minimize(qobj: qexp)`

**Arguments**
- **alg** Algorithm choice, which may be one of:
  - XPRS_BAR Newton barrier;
  - XPRS_DUAL dual simplex;
  - XPRS_LIN only solve LP ignoring all global entities;
  - XPRS_TOP stop after the LP;
  - XPRS_PRI primal simplex;
  - XPRS_GLB global search only;
  - XPRS_NIG do not call initglobal before a global search.
- **obj** Objective function constraint.
- **qobj** Quadratic objective function (with module mmquad).

**Example**
The following maximizes the objective Profit using the dual simplex algorithm and stops before the global search:
```mosel
declarations
    Profit: linctr
end-declarations

maximize(XPRS_DUAL+XPRS_TOP, Profit)
```

**Further Information**
1. These procedures call the Optimizer to maximize/minimize the current problem (excluding all hidden constraints) using the given constraint as
objective function. Optionally, the algorithm to be used can be defined. By default, the global search is executed automatically if the problem contains any global entities. Where appropriate, several algorithm choice parameters may be combined (using plus signs).

2. If XPRS_LIN is specified, then the discreteness of all global entities is ignored, even during the presolve procedure.

3. If XPRS_TOP is specified, then just the LP at the top node is solved and no Branch and Bound search is initiated. However, the discreteness of the global entities is taken into account in presolving the LP at the top node.

4. Support for quadratic programming requires the module mmquad

Related Topics
initglobal, loadprob,
**readbasis**

**Purpose**
Read a basis from a file.

**Synopsis**
procedure readbasis(fname: string, options: string)

**Arguments**
- **fname**  
  File name
- **options**  
  String of options

**Further Information**
This procedure reads in a basis from a file by calling the function ‘XPRSreadbasis’ of the optimizer. Note that basis save/read procedures can be used only if the constraint and variable names have been loaded into the optimizer (control parameter XPRS_loadnames set to true) and all constraints are named. For more detail on the options and behaviour of this procedure, refer to the Xpress-Optimizer Reference Manual.

**Related Topics**
writebasis
**readdir**

**Purpose**
Read a directives from a file.

**Synopsis**

```plaintext
procedure readdir(fname: string)
```

**Arguments**

- `fname`: File name

**Further Information**
This procedure reads in a directives from a file by calling the function ‘XPRSreaddirs’ of the optimizer. Note that directives save/read procedures can be used only if the variable names have been loaded into the optimizer (control parameter `XPRS_loadnames` set to true).

**Related Topics**
- writedirs.
savebasis

Purpose
Saves the current basis.

Synopsis
procedure savebasis(num: integer)
procedure savebasis(name: string)

Arguments
num Reference number to be given to the basis.
name Name to be given to the basis.

Further Information
This function saves the current basis giving it a reference number or a name that may subsequently be used in procedures delbasis and loadbasis.

Related Topics
delbasis, loadbasis.
setcallback

Purpose
Set Optimizer callback functions and procedures.

Synopsis
procedure setcallback(cbtype: integer, cb: string)

Arguments
cbtype  Type of the callback, which may be one of:
         XPRS_CB_LPLOG simplex log callback;
         XPRS_CB_GLOBALLOG global log callback;
         XPRS_CB_BARLOG Barrier log callback;
         XPRS_CB_CHGNODE user select node callback;
         XPRS_CB_PRENODE user preprocess node callback;
         XPRS_CB_OPTNODE user optimal node callback;
         XPRS_CB_INFNODE user infeasible node callback;
         XPRS_CB_INTSOL user integer solution callback;
         XPRS_CB_NODECUTOFF user cut-off node callback;
         XPRS_CB_INITCUTMGR cut manager initialization callback;
         XPRS_CB_FREECUTMGR cut manager termination callback;
         XPRS_CB_CUTMGR cut manager (branch and bound node) callback;
         XPRS_CB_TOPCUTMGR top node cut manager

cb  Name of the callback function/procedure. The parameters and
     the type of the return value (if any) vary depending on the type
     of the callback:
     function cb:boolean
         XPRS_CB_LPLOG, XPRS_CB_GLOBALLOG,
         XPRS_CB_BARLOG, XPRS_CB_OPTNODE,
         XPRS_CB_CUTMGR, XPRS_CB_TOPCUTMGR;
     function cb(node:integer):integer
         XPRS_CB_CHGNODE, XPRS_CB_PRENODE;
     procedure cb
         XPRS_CB_INTSOL, XPRS_CB_INFNODE,
         XPRS_CB_INITCUTMGR, XPRS_CB_FREECUTMGR;
     procedure cb(node:integer)
         XPRS_CB_NODECUTOFF.
Example

The following example defines a procedure to handle solution printing and sets it to be called whenever an integer solution is found:

```plaintext
procedure printsol
  declarations
  objval: real
  end-declarations
  objval := getparam("XPRS_lpsol")
  writeln("Solution value: ", objval)
end-procedure

setcallback(XPRS_CB_INTSOL,"printsol")
```

Further Information

This procedure sets the Optimizer callback functions and procedures. For a detailed description of these callbacks the user is referred to the Xpress-Optimizer Reference Manual. Note that whilst the solution values can be accessed from Mosel in any callback function/procedure, all other information such as the problem status or the value of the objective function must be obtained directly from the Optimizer using function `getparam`. 
setlb

**Purpose**
Set the lower bound of a variable.

**Synopsis**
procedure setlb(x: mpvar, r: real)

**Arguments**
- x A decision variable.
- r Lower bound value.

**Further Information**
This procedure changes the lower bound of a variable directly in the Optimizer, that is, the bound change is not recorded in the problem definition held in Mosel. Since this change is immediate, there is no need to reload the problem into the Optimizer (indeed, doing so resets the variable to the lower bound value computed by Mosel).

**Related Topics**
getlb, getub, loadprob, setub.
**setmipdir**

**Purpose**
Set a directive on a variable or Special Ordered Set.

**Synopsis**
- `procedure setmipdir(x: mpvar, t: integer, r: real)`
- `procedure setmipdir(x: mpvar, t: integer)`
- `procedure setmipdir(c: linctr, t: integer, r: real)`
- `procedure setmipdir(c: linctr, t: integer)`

**Arguments**
- `x` A decision variable.
- `c` A linear constraint (of type SOS).
- `r` A real value.
- `t` Directive type, which may be one of:
  - `XPRS_PR` `r` is a priority (integer between 1 and 1000, where 1 is the highest priority, 1000 the lowest);
  - `XPRS_UP` force up first;
  - `XPRS_DN` force down first;
  - `XPRS_PU` `r` is an up pseudocost;
  - `XPRS_PD` `r` is a down pseudocost.

**Further Information**
This procedure sets a directive on a global entity. Note that the priority value is converted into an integer. The directives are loaded into the Optimizer at the same time as the problem itself.

**Related Topics**
- `clearmipdir`
- `readdirs`
- `writedirs`
**setmodcut**

**Purpose**
Mark a constraint as a model cut.

**Synopsis**
```plaintext
procedure setmodcut(c: linctr)
```

**Arguments**
- `c` A linear constraint.

**Further Information**
This procedure marks the given constraint as a model cut. The list of model cuts is sent to the Optimizer when the matrix is loaded.

**Related Topics**
- `clearmodcut`. 
**setub**

**Purpose**
Set the upper bound of a variable.

**Synopsis**
```plaintext
procedure setub(x: mpvar, r: real)
```

**Arguments**
- `x`: A decision variable.
- `r`: Upper bound value.

**Further Information**
This procedure changes the upper bound of a variable directly in the Optimizer, that is, the bound change is not recorded in the problem definition held in Mosel. Since this change is immediate, there is no need to reload the problem into the Optimizer (indeed, doing so resets the variable to the upper bound value computed by Mosel). Thus the problem should be explicitly loaded with `loadprob` before setting the bound and solving.

**Related Topics**
- `getlb`, `getub`, `loadprob`, `setlb`. 
**writebasis**

**Purpose**
Write the current basis to a file.

**Synopsis**

```
procedure writebasis(fname: string, options: string)
```

**Arguments**

- `fname`  
  File name
- `options`  
  String of options

**Further Information**

This procedure writes the current basis to a file by calling the function ‘XPRSwritebasis’ of the optimizer. Note that basis save/read procedures can be used only if the constraint and variable names have been loaded into the optimizer (control parameter `XPRS_loadnames` set to true) and all constraints are named. For more detail on the options and behaviour of this procedure, refer to the Xpress-Optimizer Reference Manual.

**Related Topics**

- `readbasis`
writedirs

Purpose
Write current directives to a file.

Synopsis
procedure writedirs(fname: string)

Arguments
fname     File name

Further Information
This procedure writes the current directives to a file using the optimizer file format.

Related Topics
clearmipdir, setmipdir, readdirs.
**writeprob**

**Purpose**
Write the current problem to a file.

**Synopsis**
```plaintext
procedure writeprob(fname: string, options: string)
```

**Arguments**
- `fname` File name
- `options` String of options

**Further Information**
This procedure writes the current problem held in the optimizer to a file by calling the optimizer function ‘XPRSwriteprob’. Note that the matrix written by this procedure may be different from the one produced by `exportprob` since it may include the effects of presolve or cuts generated by the optimizer. For more detail on the options and behaviour of this procedure, refer to the *Xpress-Optimizer Reference Manual*. 
9.3 Cut Pool Manager Routines

To run the cut manager from Mosel, it may be necessary to (re)set certain Optimizer controls. For example, switching off presolve, automatic cut generation and reserving space for extra rows in the matrix may be useful:

    setparam("XPRS_presolve",0);
    setparam("XPRS_cutstrategy",0);
    setparam("XPRS_extrarows",5000);

The callback functions and procedures that are relevant to the cut manager are also initialized using the function setcallback, in common with the other Optimizer callbacks.

It should be noted that cuts are not stored by Mosel but sent immediately to the Optimizer. Consequently, if a problem is reloaded into the Optimizer, any previously defined cuts will be lost. In Mosel, cuts are defined by specifying a linear expression (i.e. an unbounded constraint) and the operator sign (inequality/equality). If instead of a linear expression a constraint is given, it will also be added to the system as an additional constraint.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
<th>Pg</th>
</tr>
</thead>
<tbody>
<tr>
<td>addcut</td>
<td>Adds a cut to the problem in the Optimizer.</td>
<td>179</td>
</tr>
<tr>
<td>addcuts</td>
<td>Adds a set of cuts to the problem in the Optimizer.</td>
<td>180</td>
</tr>
<tr>
<td>delcuts</td>
<td>Deletes cuts from the problem in the Optimizer.</td>
<td>181</td>
</tr>
<tr>
<td>dropcuts</td>
<td>Drops a set of cuts from the cut pool.</td>
<td>182</td>
</tr>
<tr>
<td>getcnlist</td>
<td>Returns the set of cuts active at the current node.</td>
<td>183</td>
</tr>
<tr>
<td>getcplist</td>
<td>Returns the set of cuts in the cut pool.</td>
<td>184</td>
</tr>
<tr>
<td>loadcuts</td>
<td>Loads cuts from the cut pool into the Optimizer.</td>
<td>185</td>
</tr>
<tr>
<td>storecut</td>
<td>Stores a cut in the cut pool.</td>
<td>186</td>
</tr>
<tr>
<td>storecuts</td>
<td>Stores multiple cuts in the cut pool.</td>
<td>187</td>
</tr>
</tbody>
</table>
addcut

Purpose
Add a cut to the problem in the Optimizer.

Synopsis
procedure addcut(cuttype: integer, type: integer,
linexp: linctr)

Arguments
cuttype Integer number for identification of the cut.
type Cut type (equation/inequality), which may be one of:
CT_GEQ inequality (‘greater than or equal to’);
CT_LEQ inequality (‘less than or equal to’);
CT_EQ equality.
linexp Linear expression (= unbounded constraint).

Further Information
This procedure adds a cut to the problem in the Optimizer. The cut is applied
to the current node and all its descendants.

Related Topics
addcuts, delcuts.
addcuts

**Purpose**

Adds an array of cuts to the problem in the Optimizer.

**Synopsis**

```plaintext
procedure addcuts(cuttype: array(range) of integer,
                 type: array(range) of integer,
                 linexp: array(range) of linctr)
```

**Arguments**

- **cuttype**  Array of integer numbers for identification of the cuts.
- **type**    Array of cut types (equation/inequality), which may be one of:
  - CT_GEQ inequality ('greater than or equal to');
  - CT_LEQ inequality ('less than or equal to');
  - CT_EQ equality.
- **linexp**   Array of linear expressions (= unbounded constraints).

**Further Information**

This procedure adds an array of cuts to the problem in the Optimizer. The cuts are applied to the current node and all its descendants. Note that the three arrays that are passed as parameters to this procedure must have the same index set.

**Related Topics**

- addcut, delcuts.
**delcuts**

**Purpose**
Delete cuts from the problem in the Optimizer.

**Synopsis**

```plaintext
procedure delcuts(keepbasis: boolean, cuttype: integer,
                  interpret: integer, delta: real, cuts: set of integer)
procedure delcuts(keepbasis: boolean, cuttype: integer,
                  interpret: integer, delta: real)
```

**Arguments**

- `keepbasis`: false cuts with nonbasic slacks may be deleted; true ensures that the basis will be valid.
- `cuttype`: Integer number for identification of the cut(s).
- `interpret`: The way in which the cut type is interpreted:
  - `-1` delete all cuts;
  - `1` treat cut types as numbers;
  - `2` treat cut types as bitmaps (delete cut if any bit matches any bit set in `cuttype`);
  - `3` treat cut types as bitmaps (delete cut if all bits match those set in `cuttype`).
- `delta`: Only delete cuts with an absolute slack value greater than `delta`. To delete all the cuts, set this argument to a very small value (e.g. `-MAX_REAL`).
- `cuts`: Set of cut indices. If not specified, all cuts of type `cuttype` are deleted.

**Further Information**
This procedure deletes cuts from the problem loaded in the Optimizer. If a cut is ruled out by any of the given criteria it will not be deleted.

**Related Topics**
addcut, addcuts.
**dropcuts**

**Purpose**
Drop a set of cuts from the cut pool.

**Synopsis**

```pascal
procedure dropcuts(cuttype: integer, interpret: integer,
cuts: set of integer)
procedure dropcuts(cuttype: integer, interpret: integer)
```

**Arguments**
- `cuttype`: Integer number for identification of the cut(s).
- `interpret`: The way in which the cut type is interpreted:
  - `-1` drop all cuts;
  - `1` treat cut types as numbers;
  - `2` treat cut types as bitmaps (delete cut if any bit matches any bit set in `cuttype`);
  - `3` treat cut types as bitmaps (delete cut if all bits match those set in `cuttype`).
- `cuts`: Set of cut indices in the cut pool. If not specified, all cuts of type `cuttype` are deleted.

**Further Information**

This procedure drops a set of cuts from the cut pool. Only those cuts which are not applied to active nodes in the branch and bound tree will be deleted.

**Related Topics**

`storecut`, `storecuts`. 
getcnlist

Purpose
Get the set of cuts active at the current node.

Synopsis
procedure getcnlist(cuttype: integer, interpret: integer,
cuts: set of integer)

Arguments
cuttype  Integer number for identification of the cut(s); \(-1\) to return all
active cuts.
interpret The way in which the cut type is interpreted:
\(-1\)  get all cuts;
\(1\)  treat cut types as numbers;
\(2\)  treat cut types as bitmaps (get cut if any bit matches
any bit set in cuttype);
\(3\)  treat cut types as bitmaps (get cut if all bits match
those set in cuttype).
cuts  Set of cut indices.

Further Information
This procedure gets the set of active cut indices at the current node in the
Optimizer. The set of cut indices is returned in the parameter cuts.

Related Topics
getcplist.
**getcplist**

**Purpose**
Get a set of cut indices from the cut pool.

**Synopsis**
```
procedure getcplist(cuttype: integer, interpret: integer, 
                   delta: real, cuts: set of integer, 
                   viol: array(range) of real)
```

**Arguments**
- **cuttype**: Integer number for identification of the cut(s).
- **interpret**: The way in which the cut type is interpreted:
  - `-1` get all cuts;
  - `1` treat cut types as numbers;
  - `2` treat cut types as bitmaps (get cut if any bit matches any bit set in `cuttype`);
  - `3` treat cut types as bitmaps (get cut if all bits match those set in `cuttype`).
- **delta**: Only return cuts with absolute slack value greater than `delta`.
- **cuts**: Set of cut indices in the cut pool.
- **viol**: Array where the slack variables for the cuts will be returned.

**Further Information**
This procedure gets a set of cut indices from the cut pool. The set of indices is returned in the parameter `cuts`.

**Related Topics**
- `getcnlist`.
loadcuts

Purpose
Load a set of cuts from the cut pool into the problem in the Optimizer.

Synopsis
procedure loadcuts(cuttype: integer, interpret: integer,
cuts: set of integer)
procedure loadcuts(cuttype: integer, interpret: integer)

Arguments
- cuttype: Integer number for identification of the cut(s).
- interpret: The way in which the cut type is interpreted:
  1. load all cuts;
  2. treat cut types as numbers;
  3. treat cut types as bitmaps (load cut if any bit matches
     any bit set in cuttype);
  4. treat cut types as bitmaps (load cut if all bits match
     those set in cuttype).
- cuts: Set of cut indices in the cut pool. If not specified, all cuts of type
  cuttype are loaded.

Further Information
This procedure loads a set of cuts into the Optimizer. The cuts remain active
at all descendant nodes.

Related Topics
storecut, storecuts.
storecut

**Purpose**
Stores a cut into the cut pool, returning its index number.

**Synopsis**

```pascal
function storecut(nodupl: integer, cuttype: integer,
                   type: integer, linexp: linctr): integer
```

**Arguments**

- `nodupl` Flag indicating how to deal with duplicate entries:
  - 0 no check;
  - 1 check for duplicates among cuts of the same cut type;
  - 2 check for duplicates among all cuts.
- `cuttype` Integer number for identification of the cut.
- `type` Cut type (equation/inequality), which may be one of:
  - `CT_GEQ` inequality (‘greater than or equal to’);
  - `CT_LEQ` inequality (‘less than or equal to’);
  - `CT_EQ` equality.
- `linexp` Linear expression (= unbounded constraint).

**Further Information**
This function stores a cut into the cut pool without applying it to the problem at the current node. The cut has to be loaded into the problem with procedure `loadcuts` in order to become active at the current node.

**Related Topics**
- `storecuts`, `loadcuts`, `dropcuts`
storecuts

Purpose
Store an array of cuts into the cut pool.

Synopsis
procedure storecuts(nodupl: integer,
                      cuttype: array(range) of integer,
                      type: array(range) of integer,
                      linexp: array(range) of linctr,
                      ndx_a: array(range) of integer)
procedure storecuts(nodupl: integer,
                      cuttype: array(range) of integer,
                      type: array(range) of integer,
                      linexp: array(range) of linctr, ndx_s: set of integer)

Arguments
nodupl Flag indicating how to deal with duplicate entries:
0 no check;
1 check for duplicates among cuts of the same cut type;
2 check for duplicates among all cuts.
cuttype Array of integer numbers for identification of the cuts.
type Array of cut types (equation/inequality), which may be one of:
CT_GEQ inequality (‘greater than or equal to’);
CT_LEQ inequality (‘less than or equal to’);
CT_EQ equality.
linexp Array of linear expressions (= unbounded constraints).
ndx_a Interval of index numbers of stored cuts.
ndx_s Set of index numbers of stored cuts.

Further Information
This function stores an array of cuts into the cut pool without applying them
to the problem at the current node. The cuts have to be loaded into the
problem with procedure loadcuts in order to become active at the current
node. The cut manager returns the indices of the stored cuts in the form of
an array ndx_a or a set of integers ndx_s. Note that the four arrays that are
passed as parameters to this procedure must have the same index set.
Related Topics
storecut, loadcuts, dropcuts
Appendix A — Syntax Diagrams for the Mosel Language

A.1 Main Structures and Statements

(Model) ::= "model" String Identifier ⟨Directives⟩ ⟨Parameters⟩ ⟨Body⟩ 'end-model' ...

(Directives) ::= ...

(Parameters) ::= ...

(Body) ::= ...

(Declarations) ::= ...

Statement}
(Type_descr) ::= ...

...

'range' (Type_name)

'range'

'set' 'of' 'integer'

'set' 'of' (Type_name)

'array' ':' (Set_def_list) 'of' (Type_name)

...
A.2 Expressions

\[\text{(Do\_block)} \text{ ::= } \text{'do'} \text{- \langle Stat\_list \rangle } \text{- 'end-do'} \text{ → }\]

\[\text{(Stat\_list)} \text{ ::= } \langle \text{Statement} \rangle \text{ → }\]

\[<\text{Init\_block}> \text{ ::= } \text{'initializations'} \text{- 'from'} \text{- \langle String\_expr \rangle } \text{- 'to'} \text{- \langle String\_expr \rangle } \text{ → }\]

\[\ldots \langle \text{Identifier} \rangle \text{- 'as'} \text{- \langle String\_expr \rangle } \text{ → }\]

\[\ldots \langle \text{Identifier} \rangle \text{- ']' } \text{- ']' } \text{ → }\]

\[\ldots \text{ 'end-initializations'} \text{ → }\]

\[\langle \text{Expression} \rangle \text{ ::= } \langle \text{Bool\_expr} \rangle \text{ → }\]

\[\langle \text{Set\_expr} \rangle \text{ → }\]

\[\langle \text{Arith\_expr} \rangle \text{ → }\]

\[\langle \text{String\_expr} \rangle \text{ → }\]

\[\langle \text{Ctr\_expr} \rangle \text{ → }\]

\[\langle \text{Comparator} \rangle \text{ ::= } \langle 'c' \rangle \text{ → }\]

\[\langle 'c=' \rangle \text{ → }\]

\[\langle '<<' \rangle \text{ → }\]

\[\langle '==' \rangle \text{ → }\]

\[\langle '>>=' \rangle \text{ → }\]

\[\langle '><=' \rangle \text{ → }\]

\[\langle '><>' \rangle \text{ → }\]

\[\langle \text{Bool\_expr} \rangle \text{ ::= } \langle \text{Bool\_expr} \rangle \text{ 'and' } \langle \text{Bool\_expr} \rangle \text{ → }\]

\[\langle 'or' \rangle \text{ → }\]

\[\langle 'and' \rangle \text{- \langle Iterator\_list \rangle } \text{- '}' } \text{- \langle Bool\_expr \rangle } \text{ → }\]

\[\langle 'or' \rangle \text{ → }\]

\[\langle 'not' \rangle \text{- \langle Expression \rangle } \text{ → }\]

\[\langle 'in' \rangle \text{- \langle Set\_expr \rangle } \text{ → }\]

\[\langle 'not' \rangle \text{- \langle Bool\_expr \rangle } \text{ → }\]

\[\langle 'if' \rangle \text{- \langle 'true' \rangle } \text{- \langle Name\_ref \rangle } \text{- 'false' } \text{ → }\]

\[\langle 'true' \rangle \text{ → }\]

\[\langle 'false' \rangle \text{ → }\]

\[\langle 'boolean' \rangle \text{- \langle Expression \rangle } \text{- '}' } \text{- \langle Bool\_expr \rangle } \text{- '}' } \text{ → }\]

\[\langle 'true' \rangle \text{- \langle Bool\_expr \rangle } \text{- '}' } \text{ → }\]

\[\langle 'false' \rangle \text{- \langle Bool\_expr \rangle } \text{- '}' } \text{ → }\]
Appendix B — Error Messages

The Mosel error messages listed in the following are grouped according to the following categories:

- **General errors**: may occur either during compilation or when running a model.
- **Parser/compiler errors**: raised during the model compilation.
- **Runtime errors**: when running a model.

All messages are identified by their code number, preceded either by the letter E for error or W for warning. Errors cause the compilation or execution of a model to fail, warnings simply indicate that there may be something to look into without causing a failure or interruption.

This chapter documents the error messages directly generated by Mosel, not the messages stemming from Mosel modules or from other libraries used by modules.

### B.1 General Errors

These errors may occur either during compilation or when running a model.

- **E-1** *Internal error in ‘location’ (errortype)*
  An unrecoverable error has been detected, Mosel exits.
  Please contact Dash.

- **E-2** *General error in ‘location’ (errortype)*
  An internal error has been detected but Mosel can recover.
  Please contact Dash.

- **E-4** *Not enough memory*
  Your system has not enough memory available to compile or execute a Mosel model.

- **E-20** *Trying to open ‘file’ twice*
  The same file cannot be opened twice (e.g. using fopen or include).

- **E-21** *I cannot open file ‘file’ for writing (operating_system_error)*
  Likely causes are an incorrect access path or write-protected files.
**E-22**  
I cannot open file ‘file’ for reading (operating_system_error)  
Likely causes are an incorrect access path or filename or not read-enabled files.

**E-23**  
Error when writing to the file ‘file’ (operating_system_error)  
The file could be opened for writing but an error occurred during writing (e.g. disk full).

**E-24**  
Error when reading from the file ‘file’ (operating_system_error)  
The file could be opened for reading but an error occurred while reading it.

**E-25**  
Unfinished string  
A string is not terminated, or different types of quotes are used to indicate start and end of a string.  
*Example:*  
```plaintext
writeln("mytext")
```

**E-26**  
Identifier expected  
May occur when reading data files: a label is missing or a numerical value has been found where a string is expected.  
*Example:*  
```plaintext
declarations
  D: range
end-declarations

initializations from "test.dat"
  D
end-initializations
```
*Contents of test.dat:*
```plaintext
[1 2 3]
The label D: is missing.
```
**E-27  Number expected**
May occur when reading data files: another data type has been found where a numerical value is expected.

*Example:*

```mosel
declarations
  C: set of real
end-declarations
initializations from "test.dat"
  C
end-initializations
```

**Contents of test.dat:**

```mosel
  C: [1 2 c]
c is not a number.
```

**E-28  Digit expected for constant exponent**
May occur when using scientific notation for real values.

*Example:*

```mosel
b := 2E -10
```

E must be immediately followed by a signed integer (i.e. no spaces).

**E-29  Wrong file descriptor number for selection (**em num**))**
*fselect* is used with an incorrect parameter value.
B.2 Parser/compiler errors

Whenever possible Mosel displays the location where an error has been detected during compilation in the format (line_number/character_position_in_line).

E-100 Syntax error before token
The parser cannot continue to analyze the source file because it has encountered an unexpected token. When the error is not an obvious syntax error, make sure you are not using an identifier that has not been defined before.
Examples:

- **token:** )
  writeln(3 mod)
  mod must be followed by an integer (or a numerical expression evaluating to an integer).

- **token:** write
  if i > 0
    write("greater")
  end-if
  then has been omitted.

- **token:** end
  if i > 0 then write("greater") end-if
  A semicolon must be added to indicate termination of the statement preceeding the end-if.

E-101 Incompatible types (type_of_problem)
We try to apply an operation to incompatible types. Check the types of the operands.

Examples:

- **type_of_problem:** assignment
  i:=0
  i:=1.5
  The first assignment defines i as an integer, the second tries to re-assign it a real value: i needs to be explicitly declared as a real.

- **type_of_problem:** cmp
  12=1=2
  A truth value (the result of 12=1) is compared to a numerical value.
E-102 *Incompatible types for parameters of ‘routine’*
A subroutine is called with the wrong parameter type. This message may also be displayed instead of E-104 if a subroutine is called with the wrong number of parameters. (This is due to the possibility to overload the definition of subroutines).

*Example:*

```mosel
procedure myprint(a:integer)
  writeln("a: ", a)
end-procedure

myprint(1.5)
```

The subroutine `myprint` is called with a real-valued argument instead of an integer.

E-103 *Incorrect number of subscripts for ‘array’(num1/num2)*
An array is used with `num2` subscripts instead of the number of subscripts `num1` indicated at its declaration.

*Example:*

```mosel
‘array’(num1/num2): ‘A’(2/1)
```

```mosel
declarations
  A: array(1..5,range) of integer
end-declarations

writeln(A(3))
```

E-104 *Incorrect number of parameters for ‘routine’(num1/num2)*
Typically displayed if `write` or `read` are used without argument(s).

E-106 *Division by zero detected*
Explicit division by 0 (otherwise error only detected at runtime).

E-107 *Math error detected on function ‘fct’*
For example, a negative number is used with a fractional exponent.

E-108 *Logical expression expected here*
Something else than a logical condition is used in an `if` statement.
E-109  Trying to redefine 'name'
      Objects can only be defined once, changing their type is not possible.
      Example:
      
      i:=0
      
      declarations
      i: real
      end-declarations
      
      i is already defined as an integer by the assignment.

E-111  Logical expression expected for operator 'op'
      Example:
      
      op: and
      2+3 and true

E-112  Numeric expression expected for operator 'op'
      Examples:
      
      op: +
      12+{13}
      
      op: *
      
      uses "mmxprs"
      
      declarations
      x:mpvar
      end-declarations

      minimize(x*x)
      
      Multiplication of decision variables of type mpvar is only possible if a
      suitable module (like mmquad) supporting non-linear expressions is
      loaded.

E-113  Wrong type for conversion
      Mosel performs automatic conversions when required (for instance from an
      integer to a real) or when explicitly requested by using the type name, e.g.
      integer(12.5). This error is raised when an unsupported conversion is
      requested or when no implicit conversion can be applied.

E-114  Unknown type for constant 'const'
      A constant is defined but there is not enough information to deduce its type
      or the type implied cannot be used for a constant (for instance a linear
      constraint).
**E-115** *Expression cannot be passed by reference*

We try to use a constant where an identifier is expected. For instance, only non-constants can be used in an initializations block.

**E-118** *Wrong logical operator*

A logical operator is used with a type for which it is not defined.

*Example:*

```
if("abc" in "acd") then writeln("?"'); end-if
```

The operator `in` is not defined for strings.

**W-121** *Statement with no effect*

An expression stands where a statement is expected. In this case, the expression is ignored - typically, a constraint has been stated and the constraint type is missing (i.e. `>=` or `<=` ...) or an equality constraint occurs without decision variables, e.g. `2=1`.

**E-122** *Control parameter ‘param’ unknown*

The control parameters of Mosel are documented in the Mosel Reference manual under function `getparam`. All control parameters provided by a module, e.g. `mmxprs`, can be display with the command "exam", e.g. `exam -c mmxprs`. In IVE this information is displayed by the module browser.

**E-123** *‘identifier’ is not defined*

*identifier* is used without or before declaring it. Check the spelling of the name. If *identifier* is defined by a module, make sure that the corresponding module is loaded. If *identifier* is a subroutine that is defined later in the program, add a forward declaration at the beginning of the model.

**E-124** *A local object may have been added to a non local set*

In the body of a subroutine, an attempt is being made to insert a local object (i.e. locally declared) into a set that may not be local to the subroutine (e.g. a parameter or a gobally declared set). This is not allowed since the local object is deleted when the subroutine terminates.

**E-125** *Set expression expected*

For instance computing the union between an integer constant and a set of integers: `union(12+(13))`.

**E-147** *Trying to interrupt a non existing loop*

*break* or *next* is used outside of a loop.
E-148 Procedure/function ‘identifier’ declared but not defined
A procedure or functions is declared with forward, but no definition of the subroutine body has been found or the subroutine body does not contain any statement.

E-150 End of file inside a commentary
A commentary (usually started with (!) is not terminated. This error may occur, for instance, with several nested commentaries.

E-151 Incompatible type for subscript num of ‘identifier’
The subscript counter num may be wrong if an incorrect number of subscripts is used.
Example:

```mosel
declarations
A:array(1..2,3..4) of integer
end-declarations

writeln(A(1.3))
```
This prints the value 2 for num, although the second subscript is actually missing.

W-152 Empty set for a loop detected
This warning will be printed in a few cases where it is possible to detect an empty set during compilation.

E-153 Trying to assign the index ‘idx’
Loop indices cannot be re-assigned.
Example:

```mosel
declarations
C: set of string
D: range
end-declarations

forall(d in D) d+=1
forall(c in C) if (c='a') then c:='A'; end-if
```
Both of these assignments will raise the error. To replace an element of the set C, the element needs to be removed and the new element added to the set.

E-154 Unexpected end of file
May occur, for instance, if an expression at the end of the model file is incomplete and in addition end-model is missing.
E-155  *Empty ‘case’*
A **case** statement is used without defining any choices.

E-156  **‘identifier’ has no type**
The type of **identifier** cannot be deduced. Typically, an undeclared object is assigned an empty set.

E-157  **Scalar expression expected**
**Examples:**

```
declarations
B=('a','b','c')
end-declarations

case B of
  1: writeln("stop")
end-case
```

The **case** statement can only be used with the basic types (integer, real, boolean, string).

```
D:=[1,2]
```

Declaration of arrays by assignment is only possible if the index set can be deduced (e.g. definition of an array of linear constraints in a loop).

E-159  **Compiler option ‘option’ unknown**
Valid compiler options include **explterm** and **noimplicit**. See the section on compiler directives in the Mosel Reference Manual for more details.

E-160  **Definition of functions and procedures cannot be nested**
May occur, for instance, if **end-procedure** or **end-function** is missing and the definition of a second subroutine follows.

E-161  **Expressions not allowed as procedure/function parameter**
Occurs typically if the index set(s) of an array are defined directly in the procedure/function prototype.
**Example:**

```
procedure myproc(F:array(1..5) of real)
  writeln("something")
end-procedure
```

Replace either by **array(range)** or **array(set of integer)** or define
```
A:=1..5 outside of the subroutine definition and use array(A)
```

E-162  **Non empty string expected here**
This error is raised, for example, by **uses ""**
**E-163** Array declarations in the form of a list are not allowed as procedure/function parameter
Basic types may be given in the form of a list, but not arrays.
*Example:*

```mosel
procedure myproc(F,G,H:array(range) of real, a,b,c:real)
  writeln("something")
end-procedure
```

**W-164** A local symbol cannot be made public
*Example:*

```mosel
procedure myproc
  declarations
    public i:integer
  end-declarations
  i:=1
end-procedure
```

Any symbol declared in a subroutine is local and cannot be made public.

**W-165** Declaration of 'identifier' hides a parameter
The name of a function/procedure parameter is re-used in a local declaration.
*Example:*

```mosel
procedure myproc(D:array(range) of real)
  declarations
    D: integer
  end-declarations
  writeln(D)
end-procedure
```

This procedure prints the value of the integer \( D \). Unless this behavior is desired, rename either the subroutine argument or the name used in the declaration.

**W-166** ';' missing at end of statement
If the option explterm is employed, then all statements must be terminated by a semicolon.
E-167 Operator ‘op’ not defined
A constructor for a type is used in a form that is not defined.
Example:
    uses "complex"
    c:=complex(1,2,3)
The module complex defines constructors for complex numbers from one
or two reals, but not from three.

E-168 ‘something’ expected here
Special case of “syntax error” (E-100) where the parser is able to provide a
guess of what is missing.
Examples:
    something :=
        a: 3
    The assignment is indicated by :=.
    something: of
        declarations
        S: set integer
        end-declarations
    of has been omitted.
    something: ...
        declarations
        A: array(1:2) of integer
        end-declarations
    Ranges are specified by ..

E-169 ‘identifier’ cannot be used as an index name (the identifier is already in use or
declared)
Example:
    i:=0
    sum(i in 1..10)
The identifier i has to be replaced by a different name in one of these
lines.

E-170 ‘=’ expects a scalar here (use ‘in’ for a set)
Special case of syntax error (E-100).
Example:
    sum(i = 1..10)
    Replace = by in.
E-171  The [upper/lower] bound of a range is not an integer expression
Example:

declarations
A: array(1..2.5) of integer
end-declarations

Ranges are intervals of integers, so the upper bound of the index range
must be changed to either 2 or 3.

Errors Related to Modules

E-302  The symbol ‘identifier’ from ‘module’ cannot be defined (redefinition)
Two different modules used by a model define the same symbol (incompatible
definitions).

E-303  Wrong type for symbol ‘identifier’ from ‘module’
Internal error in the definition of a user module (an unknown type is used):
refer to the list of type codes in the Native Interface reference manual.

W-304  The symbol ‘identifier’ is hidden by module ‘module’
Two different modules used by a model define the same symbol (definitions
are compatible, second replaces first definition).

W-306  Unknown operator ‘op’ (code num) in module ‘module’
Internal error in the definition of a user module: refer to the list of operator
codes in the Native Interface reference manual.

E-307  Operator ‘op’ (code num) from module ‘module’ rejected
Internal error in the definition of a user module: an operator is not defined
correctly.

E-308  Parameter string of a native routine corrupted
Internal error in the definition of a user module: refer to the list of parameter
type codes in the Native Interface reference manual.
B.3 Runtime Errors

Runtime errors are usually displayed without any information about where they have occurred. To obtain the location of the error, use the flag \texttt{g} with the \texttt{xpindfct(compile)}, \texttt{xpindfct(cload)}, or \texttt{xpindfct(execute)} command.

Initializations

E-30 \textit{Duplicate label 'label' at line num of file 'file' (ignored)}

The same label is used repeatedly in a data file.

\textit{Example:}

\begin{verbatim}
D: [1 2 3]
D: [1 2 4]
\end{verbatim}

E-31 \textit{Error when reading label 'label' at (num1,num2) of file 'file'}

The data entry labeled \textit{label} has not been read correctly. Usually this message is preceded by a more detailed one, e.g. E-24, E-27 or E-28.

E-32 \textit{Error when writing label 'label' at (num1,num2) of file 'file'}

The data entry labeled \textit{label} has not been written correctly. Usually this message is preceded by a more detailed one, e.g. E-23.

E-33 \textit{Initialization with file 'file' failed for: list_of_identifier}

Summary report at the end of an \texttt{initializations} section. Usually this message is preceded by more detailed ones, e.g. E-27, E-28, E-30, E-31.

General Runtime Errors

E-51 \textit{Division by zero}

Division by 0 resulting from the evaluation of an expression.

E-52 \textit{Math error performing function 'identifier'}

For example \texttt{ln} used with inadmissible argument, such as 0 or negative values.
**E-1000 Inconsistent range**
Typically displayed if the lower bound specified for a range is greater than its upper bound.

*Example:*

```mosel
D:=3..1
```

**E-1001 Conflicting types in set operation (op)**
A set operation can only be carried out between sets of the same type.

*Example:*

```mosel
declarations
C: set of integer
D: range
end-declarations

C:={5,7}
D:=C
```

The inverse, `C:=D`, is correct because ranges are a special case of sets of integers.

**E-1002 Out of range**
An attempt is being made to access an array entry that lies outside of the index sets of the array.

**E-1003 Trying to modify a finalized or fixed set**
Occurs, for instance, when it is attempted to re-assign a constant set or to add elements to a fixed set.

**E-1004 Trying to access an uninitialized object (type_of_object)**
Occurs typically in models that define subroutines.

*Example:*

```mosel
type_of_object: array
forward procedure myprint
myprint
declarations
A:array(1..2,3..4) of integer
end-declarations
procedure myprint
writeln(A(1,2))
end-procedure
```

Move the declaration of `A` before the call of the subroutine.

**E-1005 Wrong type for “procedure”**
Occurs when procedures `settype` or `getvars` are used with incorrect types.
E-1007 **Null dimension for an array**
A static index set is empty (no error raised in the case of dynamic sets).

E-1009 **Too many initializers**
The number of data elements exceeds the maximum size of an array.
*Example:*
```
declarations
  A:array(1..3) of integer
end-declarations
A:=[1,2,3,4]
```

E-1010 **Trying to extend a unary constraint**
Most types of unary constraints cannot be transformed into constraints on several variables.
*Example:*
```
declarations
  x,y: mpvar
end-declarations

c:=x is_integer
  c+:=y
```

E-1013 **Infeasible constraint**
The simple cases of infeasible unnamed constraints that are detected at run time include:
*Example:*
```
declarations
  x:mpvar
end-declarations
i:=1
if(i>=0,x,0)>=1

! or:
  x-x>=1
```

E-1100 **Empty problem**
We are trying to generate or load an empty problem into a solver (i.e. no objective function or constraints; bounds do not count as constraints).
**E-1102** Problem capacity of student license exceeded (num1 type_of_object > num2)
The problem is too large to be solved with a student license. Use a smaller data set or try to reformulate the problem to reduce the number of variables, constraints, or global entities.

**BIM Reader**

**E-80** ‘file’ is not a BIM file
Trying to load a file that does not have the structure of a BIM file.

**E-82** Wrong file version (num1/num2) for file ‘file’
A BIM file is loaded with an incompatible version of Mosel; preferably the same versions should be used for generating and running a BIM file.

**E-83** Bim file ‘file’ corrupted
A BIM file has been corrupted, e.g. by saving it with a text editor.

**E-84** File ‘file’: model cannot be renamed
A model file that is being executed cannot be re-loaded at the same time.

**W-85** Trailing data at end of file ‘file’ ignored
At the end of a BIM file additional, unidentifiable data has been found (may be a sign of file corruption).

**Module Manager Errors**

**E-350** Module ‘module’ not found
A module has not been found in the module path (directory dso of your Mosel installation, environment variable MOSEL_DSO). This message is also displayed, if a module depends on another library that has not been found (e.g. module mmxprs has been found but Xpress-Optimizer has not been installed or cannot be located by the operating system).
**E-351** *File ‘file’ is not a Mosel DSO*
Typically displayed if Mosel cannot find the module initialization function.

**E-352** *Module ‘module’: wrong interface version*
A module is not compatible with the Mosel version used to load it.

**E-353** *Module ‘module’: no authorization found*
Module *module* requires a licence for its use. Please contact Dash.

**E-354** *Error when initializing module ‘module’*
Usually preceded by an error message generated by the module. Please refer to the documentation of the module for further detail.

**E-355** *Wrong version for module ‘module’ (num1.num2/num3.num4)*
A model is run with a version of a module that is different from the version that has been used to compile the model.

**E-358** *Error when resetting module ‘module’*
A module cannot be executed (e.g. due to a lack of memory).
Index

Symbols
- 10, 20, 21
* 20, 21
+ 10, 20, 21
+= 25
, 10
/ 20
:= 25
; 11, 24
< 22
<= 22, 23
<> 22
-= 25
= 22, 23
> 22
>= 22, 23
\ 21
\n 21
_ 10

A
abs 40
absolute value 40
acces mode 144
activate
    model 5
activity 57
add
    array of cuts 180
    cut 179
addcut 41, 179
addcuts 41, 180
aggregate operator 19
and 10, 22
anonymous constraint 26
append
    file 54
arccosine 41
arcsine 41
arctan 41
arctangent function 41
arguments 33
arithmetic expression 20
array 16
    declaration 16
    dereference 18
    dynamic 16
    fixed size 16
    initialization 24, 25
array 10
as 10
assignment 24
    additive 25
    constraints 25
    subtractive 25

B
base 10 logarithm 76
basic type 15
basis
    load 162
    save 168
batch mode 3
BIM 1, 4
binary model 1, 4
bit test 42
bittest 42
boolean 10, 15, 19
Boolean expression 22
break 10, 32

C
callback 169
callback functions 178
case 10, 30
  case-insensitive 3
  case-sensitive 10
  casting 18, 19
  ceil 43
  clearmipdir 44, 154
  clearmodcut 44, 155
  clearqexpstat 136
  CLOAD 4
  close
    file 38, 50
    stream 38, 50
  coefficient 58
    set 84
  command
    operating system 3
    system 149
  command line interpreter 2, 3
  commands
    shortening 6
  comment 5, 9, 28
    sign 56
    skip 56
  commentary
    multi-line 9
  comparator 22
  COMPILE 4
  compile
    model 4
  compiled 12
  compiler directives 12
  compiler library 2
  compiler options 12
  concatenation 21
  condition 20
  connector 18
  constant 17
    declaration 17
    display 6
  constants 39
  constraint 15
    activity 57
    anonymous 26
    assignment 25
    coefficient 58
    display 6
    dual 59
    hide 85
    right hand side 23
    set coefficient 84
    set of variables 71
    set type 88
    slack 68
    type 23, 70
    constraint representation 23
    control parameter 64
      display 6
      set 86
    conversion
      basic type 19
    cos 44
    cosine function 44
    create
      directory 147
      variable 45
    create 45
    cross recursion 36
    CT_BIN 70
    CT_CONT 70
    CT_EQ 70
    CT_GEQ 70
    CT_INT 70
    CT_LEQ 70
    CT_PINT 70
    CT_SEC 70
    CT_SINT 70
    CT_SOS1 70
    CT_SOS2 70
    CT_UNB 70
    cut
      add 179
      add array 180
      delete 181
      drop 182
get active 183
cut manager 178
data display 6
declaration 12, 14
delbasis 156, 162, 168
delcuts 181
DELETE 5
delete cut 181
directory 148
deference 18
difference 21
directive 172
compiler 12
directory 142
access mode 144
create 147
delete 148
new 147
remove 148
status 144
disc 96
diskdata 97
DISPLAY 6
display info 3
model 5
models 5
symbol 6
div 10, 20
division integral 20
remainder 20
do 10
drop cut 182
dropcuts 182
DSO 6
dual value 59
dynamic 10, 17
dynamic array 16
do of variables 17
dynamic shared object constant 6
control parameter 6
examine 6
flush 6
list 6
subroutine 6
version 6
Dynamic Shared Objects manager 2
E
E-1 195
E-100 198
E-1000 208
E-1001 208
E-1002 208
E-1003 208
E-1004 208
E-1005 208
E-1007 209
elementary type  15
elif  10, 30
else  10, 30
end  10
end-case  30
end-declarations  14
end-function  35
end-if  30
end-model  12
end-procedure 35
environment variable 143
eof 72
EP_MAX 49, 129
EP_MIN 49, 129
EP_MPS 49, 129
EP_STRIP 49, 129
error
code 195
detection 12
escape sequence 21
escape sequences 21
ETC_APPEND 97
ETC_DENSE 97
ETC_IN 97
ETC_NOQ 97
ETC_NOZEROS 97
ETC_OUT 97
ETC_SGLQ 97
ETC_SPARSE 97
ETC_TRANS 97
even number 74
EXAMINE 6
execute
  model 5
exists 46
exit 47
exp 48
explterm 11, 12
exponential function 48
export
  problem 49
EXPORTPROB 5
exportprob 49, 129
expression 18
  arithmetic 20
  Boolean 22
  linear constraint 22
  print 94
  set 21
  set type 88
  string 21
terminator 11
type 18

F
F_APPEND 54
F_INPUT 54
F_OUTPUT 54
false 10, 15
fclose 38, 50
fdelete 51, 140
fflush 38, 51
file
  access mode 144
  append 54
  close 38, 50
  delete 140
  ID 60
  inclusion 13
  initialization 27
  input 38
  IO 38
  move 141
  open 38, 54
  output 38
  read 37, 81
  rename 141
  select 38, 55
  status 144
  write 37, 94
file extension 4
files
  binary model (.bim) 1, 12
finalize
  set 52
finalize 17, 52
fixed size array 16
floor 53
flush
  dynamic shared objects 6
flush buffer 51
FLUSHLIBS 6
Index 218


fmove 54, 141
fopen 38, 50, 54
forall 10, 31
format string 91
forward 10, 36
forward declaration 36
from 10
fselect 38, 55, 60
fskipline 38, 54, 56
function
  return value 34
  type 34
function 10
function call 18

G
get
  active cuts 183
  cuts from cut pool 184
getact 57
getcnalist 58, 183
getcoeff 58
getcplist 59, 184
getcwd 59, 142
getdual 59
getenv 60, 143
getfid 38, 55, 60
getfirst 61
getfstat 62, 144
getiis 157
getlast 62
getlb 63, 158
getobjval 63
getparam 54, 64
getprobstat 66, 159
getqexpnextterm 137
getqexpstat 135
getrcost 66
getsize 67
getslack 68
getsol 69, 130

H
help 3
hidden constraint 85
  constraint
    test hidden 73

I
ID
  file 60
  stream 60
identifier 10
if 10, 18, 29
implicit declaration 25
in 10, 22
include 10, 13
indexing set 16
INFO 3
initglobal 72, 161
initialisations 10
initialization 26
  file 27
initializations 10
input file 38
input stream 37, 50, 54, 55
  read 37, 81
  test eof 72
integer 10, 15, 19
integer parity 74
integral division 20
inter 10, 21
interactive mode 3
interpreted 12
intersection 21
IO
  error 54
  file 38
  status 54
  stream 38
  switching between streams 38
IOCTRL 54
IOSTATUS 54
is_binary 10, 23
is_continuous 10, 23
is_free 10, 23
is_integer 10, 23
is_partint 10, 23
is_semcont 10, 23
is_semint 10, 23
is_sos1 10, 23
is_sos2 10, 23
iseof 72
ishidden 73, 85
isodd 74
iterator 20
IVE_BLACK 101
IVE_BLUE 101
IVE_CYAN 101
IVE_GREEN 101
IVE_MAGENTA 101
IVE_RED 101
IVE_RGB 100
IVE_WHITE 101
IVE_YELLOW 101
IVEdrawarrow 102
IVEdrawlabel 103
IVEdrawline 104
IVEdrawpoint 105
IVEerase 106
IVEpause 107
IVEzoom 108

K
  keywords 6, 10

L
  language 9
    syntax diagrams 189
  largest value 78
  library
    compiler 2
    Run Time 2
  linctr 10, 15
  line breaking 10
  linear constraint expression 22
  linear expression 22
  LIST 5
  list
    dynamic shared objects 6
  ln 75
  LOAD 4
  load
    basis 162
    cut 185
    model 4
    module 12
    problem 163
  loadbasis 162, 168
  loadcuts 185, 186, 187
  loadprob 158, 160, 163
  log 76
  logarithm 76
    natural 75
  logical and 22
  logical negation 22
  logical or 22
  loop 31
  loop statement 24
  lower bound 158
    set 171
  LP format 6
  LSLIBS 6

M
  M_E 39
  M_PI 39
makedir 77, 147
makesos 77
makesos1 23
makesos2 23
matrix output 5
max 10, 20
MAX_INT 39
MAX_REAL 39
maximize 164
maximize 78, 163, 164
maximum value 20, 78
maxlist 78
min 10, 20
minimize 163
minimum value 20, 79
minlist 79
mod 10, 20
model
  activate 5
  active 5
  binary 1
  compile 4
  delete 5
  display 5
  display list 5
  execute 5
  file name 4
  load 4
  name 5
  reset 5
  run 5
  save 5
  sequence number 5
  size 5
  source 1
  structure 11
model 4, 5, 10, 12
model cut 173
model manager 2
model parameter 13
module 2
  load 12
module structure
  advantages 2
modules 9
Mosel 1
mosel 2
Mosel compiler 1
Mosel Console 2
move
  file 141
MP type 15
MPS format 6
mpvar 10, 15
multi-line commentary 9

N
natural logarithm 75
NBREAD 81
negation 22
new line 21
next 10, 32
noimplicit 12, 26
not 10, 22
not in 22
numerical constants 39

O
objective value 63
ODBC 109
  message printing 110
odd number 74
of 10
open
  file 38, 54
  stream 38, 54
operating system command 3
operation
  elementary 24
operator 18
  aggregate 19
  arithmetic 20
  optimization
direction 6
Optimizer problem status 159
option
  compiler 12
options 10, 12
or 10, 22
output
  file 38
output stream 37, 50, 54, 55
  flush 37, 51
  write 37, 94
quote 21

R
random 80, 87
random number 80, 87
range 16
range 10
range set 21
read 38, 54, 81
readln 38, 54, 81
real 10, 15, 19
recursion 33, 36
reduced cost value 66
remove
directory 148
removedir 148
rename
  file 141
repeat 10, 32
reserved words 10
RESET 5
return value 34
returned 34
round 83
rounding 43, 53, 83
RUN 5
Run Time Library 2
running time 146

S
save
basis 168
data 28
model 5
savebasis 84, 162, 168
SELECT 5
select
  file 38, 55
  stream 38, 55
selection statement 24, 29
set 16
concatenation 21
difference 21
formatted 91
get substring 93
size 67
string 10, 15, 19
string expression 21
compare 22
strip 37
subroutine 33
display 6
info 3
substr 93
sum 10, 20
summation 20
symbol
declaration 33
import 12
private 37
symbol table 37
SYMBOLS 6
syntax 9
syntax diagrams 189
SYS_DIR 144
SYS_EXEC 144
SYS_MOD 144
SYS_OTH 144
SYS_READ 144
SYS_REG 144
SYS_TYP 144
SYS_WRITE 144
SYSTEM 3
system 94, 149
system command 149
system comment 5
system status 145

T
table of symbols 37
termination 4, 47
test

bits 42
eof 72
hidden constraint 73
then 10
time measure 146
to 10
true 10, 15, 19
tuple 23
type
basic 15
constraint 23, 70, 88
conversion 19
elementary 15
MP 15
SOS 23, 70, 88
variable 23, 70, 88
type conversion 18

U
unconstrained 23
union 21
union 10, 21
unload
dynamic shared objects 6
until 10, 32
upper bound 160
set 174
user comment 5
uses 10, 12

V
variable 15
create 45
display 6
dynamic array 17
evironment 143
lower bound 158
reduced cost 66
set coefficient 84
set lower bound 171
set type 88
set upper bound 174
type 23, 70
upper bound 160
version
info 3
version number 3
VIMA 2
virtual machine interpreter 2
visual environment 3

W
W-121 201
W-152 202
W-164 204
W-165 204
W-166 204
W-304 206
W-306 206
W-85 210
warning 195
while 10, 31
working directory 142
write 37, 54, 92, 94
writeln 37, 54, 92, 94
writeprob 177

X
Xpress-IVE 3, 99
Xpress-Optimizer 2
XPRS_BAR 164
XPRS_CB_BARLOG 169
XPRS_CB_CHGNODE 169
XPRS_CB_CUTMGR 169
XPRS_CB_FREECUTMGR 169
XPRS_CB_GLOBALLOG 169
XPRS_CB_INFNODE 169
XPRS_CB_INITCUTMGR 169
XPRS_CB_INTSOL 169
XPRS_CB_LPLOG 169
XPRS_CB_NODECUTOFF 169
XPRS_CB_OPTNODE 169
XPRS_CB_PRENODE 169
XPRS_CB_TOPCUTMGR 169
XPRS_DN 172
XPRS_DUAL 164
XPRS_GLB 164
XPRS_INF 159
XPRS_LIN 164
XPRS_loadnames 152
XPRS_NIG 164
XPRS_OPT 159
XPRS_PD 172
XPRS_PR 172
XPRS_PRI 164
XPRS_PU 164
XPRS_TOP 164
XPRS_UNB 159
XPRS_UNF 159
XPRS_UP 172
XPRS_verbose 152