# Contents

## 1 Introduction
1.1 What is Mosel? ............................................. 1
1.2 General organization .................................... 1
1.3 Running Mosel ............................................ 2
   1.3.1 Command line interpreter: main commands .......... 2
   1.3.2 Command line interpreter: debugger ................. 7
1.4 References .............................................. 8
1.5 Structure of this manual ................................ 8

## I Core System

## 2 The Mosel Language
2.1 Introduction ............................................ 10
   2.1.1 Comments ........................................... 10
   2.1.2 Identifiers ......................................... 10
   2.1.3 Reserved words .................................... 11
   2.1.4 Separation of instructions, line breaking .......... 11
   2.1.5 Conventions in this document ......................... 11
2.2 Structure of the source file ............................ 11
2.3 The compiler directives ................................. 12
   2.3.1 Directive uses ...................................... 12
   2.3.2 Directive imports .................................. 13
   2.3.3 Directive options .................................. 13
   2.3.4 Directive version ................................... 14
2.4 The parameters block ................................... 14
2.5 Source file preprocessing ............................... 14
   2.5.1 Source file inclusion ................................ 14
   2.5.2 Line control directives .............................. 15
2.6 The declaration block .................................. 15
   2.6.1 Elementary types ................................... 16
      2.6.1.1 Basic types ..................................... 16
      2.6.1.2 MP types ....................................... 16
   2.6.2 Sets ................................................ 16
   2.6.3 Lists ................................................ 17
   2.6.4 Arrays ............................................... 17
      2.6.4.1 Special case of dynamic arrays of a type not supporting assignment 18
   2.6.5 Records ............................................. 18
   2.6.6 Constants .......................................... 19
   2.6.7 User defined types ................................ 19
      2.6.7.1 Naming new types ............................... 19
      2.6.7.2 Combining types ................................ 20
2.7 Expressions ............................................. 20
   2.7.1 Introduction ........................................ 20
   2.7.2 Aggregate operators ................................ 21
   2.7.3 Arithmetic expressions ............................. 22
   2.7.4 String expressions ................................. 22


<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.7.5</td>
<td>Set expressions</td>
<td>23</td>
</tr>
<tr>
<td>2.7.6</td>
<td>List expressions</td>
<td>23</td>
</tr>
<tr>
<td>2.7.7</td>
<td>Boolean expressions</td>
<td>23</td>
</tr>
<tr>
<td>2.7.8</td>
<td>Linear constraint expressions</td>
<td>24</td>
</tr>
<tr>
<td>2.8</td>
<td>Statements</td>
<td>25</td>
</tr>
<tr>
<td>2.8.1</td>
<td>Simple statements</td>
<td>25</td>
</tr>
<tr>
<td>2.8.1.1</td>
<td>Assignment</td>
<td>25</td>
</tr>
<tr>
<td>2.8.1.2</td>
<td>Assignment of structured types</td>
<td>25</td>
</tr>
<tr>
<td>2.8.1.3</td>
<td>About implicit declarations</td>
<td>26</td>
</tr>
<tr>
<td>2.8.1.4</td>
<td>Inline initialization</td>
<td>26</td>
</tr>
<tr>
<td>2.8.1.5</td>
<td>Linear constraint expression</td>
<td>27</td>
</tr>
<tr>
<td>2.8.1.6</td>
<td>Procedure call</td>
<td>27</td>
</tr>
<tr>
<td>2.8.2</td>
<td>Initialization block</td>
<td>27</td>
</tr>
<tr>
<td>2.8.3</td>
<td>Selections</td>
<td>29</td>
</tr>
<tr>
<td>2.8.3.1</td>
<td>If statement</td>
<td>29</td>
</tr>
<tr>
<td>2.8.3.2</td>
<td>Case statement</td>
<td>30</td>
</tr>
<tr>
<td>2.8.4</td>
<td>Loops</td>
<td>30</td>
</tr>
<tr>
<td>2.8.4.1</td>
<td>Forall loop</td>
<td>30</td>
</tr>
<tr>
<td>2.8.4.2</td>
<td>While loop</td>
<td>31</td>
</tr>
<tr>
<td>2.8.4.3</td>
<td>Repeat loop</td>
<td>31</td>
</tr>
<tr>
<td>2.8.4.4</td>
<td>break and next statements</td>
<td>31</td>
</tr>
<tr>
<td>2.9</td>
<td>Procedures and functions</td>
<td>32</td>
</tr>
<tr>
<td>2.9.1</td>
<td>Definition</td>
<td>32</td>
</tr>
<tr>
<td>2.9.2</td>
<td>Formal parameters: passing convention</td>
<td>33</td>
</tr>
<tr>
<td>2.9.3</td>
<td>Local declarations</td>
<td>33</td>
</tr>
<tr>
<td>2.9.4</td>
<td>Overloading</td>
<td>33</td>
</tr>
<tr>
<td>2.9.5</td>
<td>Forward declaration</td>
<td>34</td>
</tr>
<tr>
<td>2.9.6</td>
<td>Suffix notation</td>
<td>34</td>
</tr>
<tr>
<td>2.10</td>
<td>The public qualifier</td>
<td>35</td>
</tr>
<tr>
<td>2.11</td>
<td>Packages</td>
<td>35</td>
</tr>
<tr>
<td>2.11.1</td>
<td>The requirements block</td>
<td>36</td>
</tr>
<tr>
<td>2.12</td>
<td>File names and input/output drivers</td>
<td>36</td>
</tr>
<tr>
<td>2.13</td>
<td>Handling of input/output</td>
<td>36</td>
</tr>
<tr>
<td>3</td>
<td>Predefined functions and procedures</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>abs</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>arctan</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>assert</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>bittest</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>ceil</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>cos</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>create</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>exists</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td>exit</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>exp</td>
<td>48</td>
</tr>
<tr>
<td></td>
<td>exportprob</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>currentdate</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>currenttime</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td>cuthead</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>cuttail</td>
<td>53</td>
</tr>
<tr>
<td></td>
<td>delcell</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>fclose</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>fflush</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>finalize</td>
<td>57</td>
</tr>
<tr>
<td></td>
<td>findfirst</td>
<td>58</td>
</tr>
<tr>
<td></td>
<td>findlast</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>floor</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>fopen</td>
<td>61</td>
</tr>
<tr>
<td></td>
<td>fselect</td>
<td>62</td>
</tr>
</tbody>
</table>

---

Contents
fspanline ................................................................. 63
getact ................................................................. 64
getcoeff ............................................................... 65
getdual ................................................................. 66
getfid ................................................................. 67
getfname .............................................................. 68
getfirst ............................................................... 69
gethead ................................................................. 70
getlast ................................................................. 71
getobjval ............................................................. 72
getparam ............................................................. 73
getcost ................................................................. 74
getreadcnt .......................................................... 75
getreverse ............................................................ 76
getsize ................................................................. 77
getslack ............................................................... 78
getsol ................................................................. 79
gettype ............................................................... 80
gettail ................................................................. 81
getvars ................................................................. 82
iseof ................................................................. 83
ishidden ............................................................... 84
isodd ................................................................. 85
In ................................................................. 86
log ................................................................. 87
makesos1, makesos2 ................................................ 88
maxlist ................................................................. 89
minlist ................................................................. 90
random ............................................................... 91
read, readln ......................................................... 92
reverse ............................................................... 93
round ................................................................. 94
setcoeff .............................................................. 95
sethidden ............................................................ 96
setparam ............................................................. 97
setrandseed ........................................................ 98
settype ............................................................... 99
sin ................................................................. 100
splithead .............................................................. 101
splittail ............................................................. 102
sqrt ................................................................. 103
strfmt ............................................................... 104
substr ............................................................... 105
timestamp .......................................................... 106
write, writeln ....................................................... 107

II Modules

4 mmetc .............................................................. 109
  4.1 Procedures and functions .................................... 109
      disc ............................................................ 110
      diskdata ....................................................... 111
  4.2 I/O drivers .................................................... 112
      4.2.1 Driver diskdata ........................................ 112

5 mmive ............................................................ 113
  5.1 Procedures and Functions .................................... 113
      IVE_RGB ....................................................... 114
6 mmjava
   6.1 I/O drivers .................................................. 126
      6.1.1 Driver java ........................................... 126
      6.1.2 Driver jraw ........................................... 127

7 mmjobs
   7.1 Example .................................................... 128
   7.2 Procedures and functions
      7.2.1 Model management
         compile ...................................................... 129
         load ....................................................... 130
         setdefstream ............................................... 131
         run ........................................................... 132
         getid ........................................................ 133
         getstatus ................................................... 134
         getexitcode ............................................... 135
         stop ........................................................ 136
         reset ......................................................... 137
         unload ....................................................... 138
      7.2.2 Synchronization
         send ........................................................ 139
         wait ........................................................ 140
         waitfor ..................................................... 141
         getnextevent ............................................... 142
         dropnextevent ............................................. 143
         isqueueempty ............................................... 144
         nullevent .................................................. 145
         getfromid .................................................. 146
         getclass .................................................... 147
         getvalue .................................................... 148
         nullevent .................................................. 149
         getstatus ................................................... 150
      7.3 I/O drivers ................................................ 151
         7.3.1 Driver shmec .......................................... 151
         7.3.2 Driver mempipe ...................................... 151

8 mmoci
   8.1 Example .................................................... 153
   8.2 Data transfer between Mosel and Oracle
      8.2.1 From Oracle to Mosel ................................ 154
      8.2.2 From Mosel to Oracle ................................ 155
   8.3 Control parameters
      OCIautocommit ............................................. 156
      OCIlbufsize ................................................ 157
      OCIcolsz ................................................... 157
      OCIconnection .............................................. 158
      OCIddebug .................................................. 158
      OCIdxcol .................................................... 158
      OCIrowcnt .................................................. 159
Contents

8.4 Procedures and functions
  OCIrowxfr ................................................................. 159
  OCIsuccess ................................................................. 159
  OCIverbose ................................................................. 159

9 mmodb
  9.1 Prerequisite .......................................................... 170
  9.2 Example ................................................................. 170
  9.3 Data transfer between Mosel and the database
    9.3.1 From the database to Mosel .................................. 171
    9.3.2 From Mosel to the database .................................. 172
  9.4 ODBC and MS Excel .................................................. 173
  9.5 Control parameters .................................................. 174
    SQLbufsize ............................................................... 175
    SQLcolsize ............................................................... 175
    SQLconnection ........................................................ 175
    SQLdebug ................................................................. 175
    SQlDM ...................................................................... 176
    SQLextn ................................................................. 176
    SQLndxcol ............................................................... 176
    SQLrowcnt .............................................................. 176
    SQLrowxfr ............................................................... 177
    SQLsuccess .............................................................. 177
    SQLverbose .............................................................. 177
  9.6 Procedures and functions .......................................... 177
    SQLconnect .............................................................. 178
    SQLdisconnect ......................................................... 179
    SQLexecute .............................................................. 180
    SQLreadinteger ......................................................... 181
    SQLreadreal ............................................................. 182
    SQLreadstring .......................................................... 183
    SQLupdate ............................................................... 184
  9.7 I/O drivers ............................................................ 185
    9.7.1 Driver odbc ...................................................... 185
    9.7.2 Driver excel .................................................... 185

10 mmquad
  10.1 New functionality for the Mosel language
    10.1.1 The type qexp and its operators
      10.1.1.1 Example: using mmquad for Quadratic Programming ... 187
    10.1.2 Procedures and functions
      exportprob ............................................................ 189
      getsol ................................................................. 190
  10.2 Published library functions ..................................... 191
    10.2.1 Complete module example ................................... 191
    10.2.2 Description of the library functions
      getqexpsol ............................................................ 193
      getqexpstat .......................................................... 195
      clearqexpstat ....................................................... 196
11 mmSystem

11.1 New functionality for the Mosel language

11.1.1 The type text ........................................ 198
11.1.2 The type date ........................................ 198
11.1.3 The type time ........................................ 198
11.1.4 The type datetime ................................. 199

11.2 Control parameters

datefmt ........................................ 199
timefmt ........................................ 200
datettimefmt ..................................... 200
monthnames ....................................... 200
sys_endparse ...................................... 201
sys_fillchar ...................................... 201

11.3 Procedures and functions

fcopy ........................................ 203
fdelete ......................................... 204
fmove ........................................... 205
getcwd ........................................... 206
getenv ........................................... 207
setenv ........................................... 208
getfstat ......................................... 209
getfsize ......................................... 210
getftime ......................................... 211
getsysstat ....................................... 212
gettime ........................................... 213
getdate ........................................... 214
makedir ......................................... 215
openpipe ......................................... 216
qsort ............................................. 217
removedir ....................................... 218
sleep ............................................. 219
system ............................................ 220
getday ............................................ 221
getmonth ......................................... 222
getyear ........................................... 223
getweekday ....................................... 224
gethour ........................................... 225
getminute ......................................... 226
getsecond ......................................... 227
getmsec ........................................... 228
getasnumber ...................................... 229
isvalid ........................................... 230
setday ............................................. 231
setmonth ......................................... 232
setyear ........................................... 233
sethour ........................................... 234
setminute ......................................... 235
setsecond ......................................... 236
setmsec ........................................... 237
settime ........................................... 238
setdate ........................................... 239
textfmt ............................................ 240
gettextsize ....................................... 241
copytext ........................................... 242
cuttext ............................................ 243
deltxt ............................................. 244
pastetext ......................................... 245
Contents

12 mmxprs  259
12.1 New functionality for the Mosel language  259
12.1.1 The type basis  259
12.2 Control parameters  259
   XPRS_colorder  260
   XPRS_loadnames  260
   XPRS_problem  260
   XPRS_probnames  260
   XPRS_verbose  260
12.3 Procedures and functions  261
   clearmipdir  263
   clearmodcut  264
   command  265
   defsecurevecs  266
   stopoptimize  267
   fixglobal  268
   getbstat  269
   getiis  270
   getinefeas  271
   getlb  272
   getname  273
   getprobstat  274
   getrange  275
   getsensrng  276
   getub  277
   initglobal  278
   isintegral  279
   loadbasis  280
   loadmipsol  281
   loadprob  282
   unloadprob  283
   maximize, minimize  284
   readbasis  285
   readdirs  286
   resetbasis  287
   savebasis  288
   savemipsol  289
   savestate  290
   setbstat  291
   setcallback  292
   setlb  294
   setmipdir  295
   setmodcut  296

11.4 I/O drivers  254
11.4.1 Driver pipe  254
11.5 Published library functions  254
   11.5.1 Description of the library functions  255
   gettxtsize  256
   gettxtbuf  257
   txtresize  258

inserttext  246
findtext  247
getchar  248
setchar  249
readtextline  250
parseint  251
parsereal  252
parseextn  253

12 mmxprs  259
Chapter 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. An interactive debugger as well as a profiler are also included: the debugger allows to execute the model step by step, specify breakpoints from where status of the model can be examined. Running a model with the profiler provides detailed information on what part of the code is actually executed and how much time each statement requires. This information may be helpful for optimizing the model (by locating hot spots where the code is using a great deal of computer time) and for building testsuites for instance (by checking whether the data sets used in the test set exercise all statements of a given model).

1.3.1 Command line interpreter: main commands

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,

mosel -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:

> system ls
> system vi mywork.mos

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.

SETSTREAM [-i|o|e] filename [file_out file_err]: Set one (or all) of the default streams: without option, three filenames are required (input, output and error stream). Note that if one of these file names is ‘*‘, the corresponding stream is not modified. A single stream may be updated using one of the option flags (‘-i‘ for input, ‘-o‘ for output and ‘-e‘ for error). If no model is loaded, the setting becomes the default for models loaded later and the system switches to the given error stream (for error messages during compilation for instance). Otherwise the change applies to the current model.

QUIT  Terminate the current Mosel session.

COMPILE [-sgGp] filename [comment [dst_file]]  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. If option dst_file (filename to use for saving BIM file) is not given, the extension .bim is used to form the name of the binary file. 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 flag ‘-G‘ adds both debugging and tracing information: it is required to run the model with the debugger. When this flag is used, the compiler adds instructions in the generated code that may slow down execution speed of the model. The optional comment parameter may be used to add a commentary to the BIM file (cf. command LIST). Note that the source file name may contain environment variable expansions using the notation ${varname} (e.g. '${MOSEL}/source/model'). When this facility is used, an output file name must be provided. 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 [-sgG] 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 and version number (given by the model and version statements 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 (cf. COMPILE, CLOAD);
- modules: the name and version number of each module required by the model;
- pkg. req.: if the model is a package, the name and version number of each package required by a model using this package;
- pkg. imp.: the name and version number of each package included by this model.

The active model is marked by an asterisk (‘*’) in front of its name (the commands UNLOAD, 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.

UNLOAD [number | name]  Unload 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 unloaded. 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. Pressing ctrl-C causes the execution to be canceled. Note that if a critical operation is being processed, the interruption is delayed until the operation completes. For instance, the Optimizer cannot be interrupted during an iteration of its algorithm. That Examples:
DEBUG [parameters] Execute the active model with the debugger (cf. Section 1.3.2). 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. This command requires that the model has been compiled with options -g or -G.

PROFILE [-sp] [parameters] Execute the active model with the profiler. 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. This command requires that the model has been compiled with options -G. After execution, the total execution time and some source coverage information is displayed. Moreover, if options -s has not been specified, a file sourcefile.prof is generated based on the original source file. Each line of this file consists in:

- the number of times the corresponding statement has been executed;
- the total amount of time (in seconds) or the percentage of the total execution time (if option -s is used) spent on this particular line (this measure is not valid if the statement is a recursive call);
- the elapsed time (in seconds) between the beginning of the execution and the last time the line was executed;
- the text of the model source

All lines of the original source file are transferred, lines not corresponding to the beginning of a statement are directly copied without further information.

EXEC [-sg] 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 Re-initialize 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.

PRINT expression [xfilename] Evaluate then display the value of the given arithmetic or Boolean expression. For building the expression, the following functions can be used: getparam, ceil, floor, round, abs, getsol, getsize, getrcost, getdual, getslack, getact. get-functions may be called using the suffix notation (e.g. getact(c) is equivalent to c.act). Some functions can be applied to arrays: the result is the evaluation of the function for each cell of the array. Private symbols of packages may be accessed by prefixing the symbol name by the package name and the symbol ‘˜’ (for instance the identifier aa declared in the package mypkg can be accessed using mypkg˜aa). Display format of this command is compatible with the data file format of Mosel. Use operator »filename to append output of the command to file filename.

Examples:

>print getsol(x) >> solfile.txt
>print getact(C(1,"tut")+c.size
>print totot-a
>print abs(mytol)>1
OPTION name [ (= ) value] View or change the value of a command line parameter. These parameters are used by the command line interpreter to display real values (especially in command PRINT):

- realfmt: C-style format for printing floating point numbers (default value: "%g")
- zerotol: zero tolerance to decide whether two values are equal (default value: 1e-13). It is also used when printing very small numbers: if a value is smaller than zerotol, "0" is displayed instead.

Although these parameters have same name and function than those used by Mosel when running a model, they are not synchronised with their internal counterpart.

SYMBOLS [-cspou] 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. By default the list is sorted in alphabetical order, option ‘-u’ disables sorting.

LSDRVS Display the list of IO drivers currently available.

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 [-cspdtu] libname Display the list of constants, procedures/functions, types, IO drivers 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, ‘-d’ for IO drivers and ‘-p’ for control parameters. By default listings are sorted in alphabetical order, option ‘-u’ disable sorting.

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 e command). String arguments (the parameter 10 is a number, but "10" or ’10’ are text strings) 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.

The mosel command terminates with the following exit status:

0: Normal termination
1: Unrecoverable error
2: Execution interrupted (ctrl-C, compilation failed...)
3: License error
4: Memory error

Moreover, if run or exec is the last command executed by the interpreter, the value returned is the one provided via procedure exit in the model (by default this is 0).

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

```plaintext
> cload mymodel
> run
```

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

```plaintext
> exec model
```
1.3.2 Command line interpreter: debugger

When a model executed through the debugger is interrupted (because the user has typed ctrl-C or an error occurred for instance), the execution is suspended, the command line interpreter displays the text source of the statement being processed and enters debug mode. In this mode, the normal command prompt is replaced by ‘>dbg’ and only commands EXPORTPROB, INFO, OPTION and PRINT are still available. In addition the following commands are enabled:

CONTINUE Resume execution. If the interruption was not due to an error, execution of the model continues, otherwise the execution of the model is aborted and the command line interpreter returns to its normal mode.

STEP Continue execution until the next statement stepping into procedures and functions. The execution continues but will be interrupted again after the current statement has been completed. If the current statement contains function or procedure calls, interruption will happen in these procedures or functions.

NEXT [line [file]] Continue execution until the next statement. The execution continues but will be interrupted again after the current statement has been completed. If a location information is provided (by means of a line number and, if necessary, a file name), the next interruption will occur before the specified statement is executed.

QUIT Terminate the debug session. Execution is aborted and the command line interpreter returns to its normal mode.

LIST [[start] nblines] Display the source file corresponding to the model being executed. When used with no extra argument, this command lists 10 lines of the source model starting at the current statement; used with a single positive parameter nblines, it displays nblines lines instead of the default 10 lines. If the parameter nblines is negative, it is interpreted as a starting point for the listing relative to the current statement. When 2 parameters are used, the first one is understood as the first line to display (a negative value is relative to the current line) and the second one as the number of lines to display.

Examples (assuming current line is 5):

- >list displays lines 5 to 14
- >list 5 displays lines 5 to 9
- >list -2 displays lines 3 to 14
- >list -2 5 displays lines 3 to 7

DISPLAY [expression] Record an expression to be displayed at each interruption. Used with no expression, this command gives a list of all recorded expressions.

UNDISPLAY [disp] Remove an expression recorded with DISPLAY. If no parameter is provided, all recorded expressions are removed otherwise the parameter is understood as a record number.

BREAK [procname][[line [file]]] Install a breakpoint. When a breakpoint has been set up, execution is interrupted whenever the statement corresponding to the specified location is reached. A procedure or function name may be used as the location: in this case a breakpoint is installed at the beginning of each procedure or function of the provided name. If used with no parameter, the breakpoint is defined at the current location.

BCONDITION bk [cond] Define or remove a condition on a breakpoint. This command may be used to put a condition (Boolean expression) on the specified breakpoint: the execution is suspended at the breakpoint only if the given condition is verified. To remove a condition previously set up, the command should be typed without specifying the condition.

DELETE [bk] Delete a breakpoint.

BREAKPOINTS List defined breakpoints.
WHERE [nblev] Display the calling stack. The calling stack corresponds to the sequence of procedure and function calls being processed. For instance assume the model calls procedure A which calls procedure B and the execution is suspended in procedure B: the calling stack will contain 3 records (location where A is called, location where B is called and current statement).

UP [nblev] Go up in the calling stack. If an argument is provided, it indicates how many levels up to go (default is 1). Note that expressions are evaluated according to the current stack frame. For instance is variable \( i \) is defined in procedure B and execution is suspended in procedure A called by B, it is necessary to go up in the stack in order to view the value of \( i \) because it does not exist in the current frame.

DOWN [nblev] Go down in the calling stack. If an argument is provided, it indicates how many levels down to go (default is 1).

Execution step by step and breakpoints can be used only if the model has been compiled using option \(-G\). In this case, before the execution starts, a breakpoint is automatically put at the first statement of the model. Otherwise (model has been compiled with option \(-g\)), the model will be interrupted only if an error occurs or keys ctrl-C are pressed.

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 basic building blocks of Mosel’s modeling and programming language are discussed.

Chapter 3 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.
I. Core System
Chapter 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.

2.1 Introduction

2.1.1 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 line 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 analyzed
(! Start of a multi line
|(! Another comment
| blabla
| end of the second level comment !)
| end of the first level !) Analysis continues here

Comments may appear anywhere in the source file.

2.1.2 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’).
2.1.3 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, dynamic, elif, else, end, evaluation, false, forall, forward, from, function, if, imports, 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, list, max, min, mod, model, mivar, next, not, of, options, or, package, parameters, procedure, public, prod, range, real, record, repeat, requirements, set, string, sum, then, to, true, union, until, uses, version, while.

Note that, although the lexical analyzer 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).

2.1.4 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 analyzer that the expression continues in the following line(s).

A+B   ! Expression 1
-A+C+D ! Expression 2
A+B-   ! Expression 3...
C+D    ! ...end of expression 3

Moreover, the character ‘;’ can be used as an expression terminator.

A+B ! Expression 1
-A+C+D ! 2 expressions on the same line
A+B ! Expression 3...
-A+C+D ! ...end of expression 3

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) 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 ‘;’.

A+B;   ! Expression 1
-A+C+D ! Expression 2
A+B    ! Expression 3...
-A+C+D ! ...end of expression 3

2.1.5 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 Mosel compiler may compile both models and packages source files. Once compiled, a
model is ready for execution but a package is intended to be used by a model or another package (see Section 2.3).

The general structure of a model 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.

The structure of a package (see Section 2.11) source file is similar to the one of a model:

```mosel
package package_name
[ Directives ]
[ Parameters ]
[ Body ]
end-package
```

The `package` statement marks the beginning the library and the statement `end-package` its end. The package name must be a valid identifier.

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 four different types of directives: the `uses` statement, the `imports` statement, the `options` statement and the `version` statement.

#### 2.3.1 Directive uses

The general form of a `uses` statement is:

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

This clause asks the compiler to load the listed modules or packages and import the symbols they define. Modules must still be available for running the model but packages are incorporated into the generated bim file when compiling a model. If the source file being processed is a package, the bim files associated to the listed packages must be available for compiling another file using this package. It is also possible to merge bim files of several packages by using `imports` instead of `uses` when building packages.
By default the compiler tries first to find a package (the corresponding file is \textit{libname.bim}) then, if this fails, it searches for a module (which file name is \textit{libname.dso}). It is possible to indicate the type of library to look for by appending either \texttt{.bim} or \texttt{.dso} to the name (then the compiler does not try the alternative in case of failure). A package may also be specified by an extended file name (see Section 2.12) including the IO driver in order to disable the automatic search (\textit{i.e.} \texttt{"a.bim"} searches the file \texttt{a.bim} in the library path but \texttt{":a.bim"} takes the file \texttt{a.bim} from the current directory).

For example,

\begin{verbatim}
uses 'mmsystem','mmxprs.dso','mypkg.bim'
uses ':/tmp/otherpkg.bim'
\end{verbatim}

Both packages and modules are searched in a list of possible locations initialized by means of environment variables. Upon startup, Mosel uses as the default for this list the value of the environment variable MOSEL_DSO completed by the subdirectory \texttt{dso} in the Mosel installation directory. This directory is taken from one of the environment variables MOSEL, XPRESSDIR or XPRESS (note that Mosel console still works even if none of these variables is defined). The variable \texttt{MOSEL_DSO} is expected to be a list of paths conforming to the operating system conventions: for a Unix system the path separator is \	exttt{"\textasciitilde"} (\textit{e.g.} \	exttt{"/opt/Mosel/dso:/tmp"}) and it is \	exttt{";"} under Win32 (\textit{e.g.} \	exttt{"E:\Mosel\Dso;C:\Temp"}). The search path for modules and packages may also be inspected and modified from the Mosel Libraries (see functions \texttt{XPRMgetdsopath} and \texttt{XPRMsetdsopath} in the \textit{Mosel Libraries Reference Manual}).

\subsection*{2.3.2 Directive \texttt{imports}}

The general form of an \texttt{imports} statement is:

\begin{verbatim}
imports pkgname1 [, pkgname2 ...];;
\end{verbatim}

This clause is a special version of the \texttt{uses} directive that can only be used in packages: it asks the compiler to load the listed packages, import the symbols they define and incorporate the corresponding bim file. As a consequence, the generated package provides the functionality of the packages it imports.

For example,

\begin{verbatim}
imports 'mypkg'
\end{verbatim}

\subsection*{2.3.3 Directive \texttt{options}}

The compiler options may be used to modify the default behaviour of the compiler. The general form of an \texttt{options} statement is:

\begin{verbatim}
options optname1 [, optname2 ...];
\end{verbatim}

The supported options are:

- \texttt{explterm}: asks the compiler to expect explicit expression termination (see Section 2.1.4)
- \texttt{noimplicit}: disables the implicit declarations (see Section 2.8.1.3)
- \texttt{keepassert}: assertions (\textit{cf. assert}) are compiled only in debug mode. With this option assertions are preserved regardless of the compilation mode.

For example,

\begin{verbatim}
options noimplicit,explterm
\end{verbatim}
2.3.4 Directive version

In addition to the model/package name, a file version number may be specified using this
directive: a version number consists in 1, 2 or 3 integers between 0 and 999 separated by the
character ‘.’.

\[ \text{version major [. minor [. release ]] } \]

For example,

\[ \text{version 1.2} \]

The file version is stored in the BIM file and can be displayed from the Mosel console (command
list) or retrieved using the Mosel Libraries (see function XPRMgetmodprop in the Mosel Li-
braries Reference Manual). From the model itself, the version number is recorded as a string
in the control parameter mosel_version (see function getparam).

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:

\[
\begin{align*}
\text{parameters} \\
\quad \text{ident1} = \text{Expression1} \\
\quad [ \text{ident2} = \text{Expression2} \ldots ] \\
\text{end-parameters}
\end{align*}
\]

where each identifier \( \text{identi} \) is the name of a parameter and the corresponding expression
\( \text{Expressioni} \) its default value. This value is assigned to the parameter if no explicit value is pro-
vided 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).

\[
\text{parameters} \\
\quad \text{size}=12 \quad ! \text{Integer parameter} \\
\quad \text{R}=12.67 \quad ! \text{Real parameter} \\
\quad \text{F}="\text{myfile}" \quad ! \text{Text string parameter} \\
\quad \text{B}=\text{true} \quad ! \text{Boolean parameter} \\
\text{end-parameters}
\]

In addition to model parameters, Mosel and some modules provide control parameters : they
can be used to give information on the system (e.g. success of an I/O operation) or control
its behaviour (e.g. select output format of real numbers). These parameters can be accessed
and modified using the routines getparam and setparam. Refer to the documentation of
these functions for a complete listing of available Mosel parameters. The documentation of
the modules include the description of the parameters they publish.

2.5 Source file preprocessing

2.5.1 Source file inclusion

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

\[ \text{include filename} \]
where *filename* is the name of the file to be included. This file name may contain environment variable references using the notation \${varname} (e.g. \`${MOSEL}/examples/mymodel\`) that are expanded to generate the actual name. The ‘include’ instruction is replaced at compile time by the contents of the file *filename*.

Assuming the file *a.mos* contains:

```mosel
model "Example for file inclusion"
write('From the main file')
include "b.mos"
end-model
```

And the file *b.mos*:

```mosel
write('From an included file')
```

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

```mosel
model "Example for file inclusion"
write('From the main file')
write('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*).

### 2.5.2 Line control directives

In some cases it may be useful to process a Mosel source through an external preprocessor before compilation. For instance this may enable the use of facilities not supported by the Mosel compiler like macros, unrestricted file inclusion or conditional compilation. In order to generate meaningful error messages, the Mosel compiler supports *line control* directives: these directives are inserted by preprocessors (e.g. cpp or m4) to indicate the original location (file name and line number) of generated text.

```mosel
#\[/line] linenum \[filename]\[
```

To be properly interpreted, a line control directive must be the only statement of the line. Malformed directives and text following valid directives are silently ignored.

### 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 organized, 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*.

```mosel
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.
2.6.1 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:

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

where \text{type\_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 (\text{integer, real, string, boolean}), an MP type (\text{mpvar, linctr}), an external type or a user defined type (see section 2.6.7). 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).

\[
\text{declarations}
\begin{align*}
  i,j: \text{integer} \\
  \text{str: string} \\
  x,y,z: \text{mpvar}
\end{align*}
\text{end-declarations}
\]

2.6.1.1 Basic types

The basic types are:

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

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

2.6.1.2 MP types

Two special types are provided for mathematical programming.

- \text{mpvar}: a decision variable
- \text{linctr}: a linear constraint

2.6.2 Sets

Sets are used to group an unordered collection of elements of a given type. Set elements are unique: if an element is added several times it is only contained once in the set. Declaring a set consists of defining the type of elements to be collected.

The general form of a set declaration is:

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

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 an ordered collection of consecutive integers in a given interval. The declaration of a range set is achieved by:

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

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

\begin{verbatim}
declarations
 sl: set of string
 rl: range
end-declarations
\end{verbatim}

### 2.6.3 Lists

Lists are used to group a collection of elements of a given type. An element can be stored several times in a list and order of the elements is specified by construction. Declaring a list consists of defining the type of elements to be collected.

The general form of a list declaration is:

\[ \text{ident1 [, ident2 ...]: list of type_name} \]

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

Every newly created list is empty.

\begin{verbatim}
declarations
 l1: list of string
 l2: list of real
end-declarations
\end{verbatim}

### 2.6.4 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 characterized 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 explicitly (cf. procedure \text{create}) or when they are assigned a value (cf. Section 2.8.1.1) and the array may then grow ‘on demand’. It is also possible to delete some or all cells of a dynamic array using the procedure \text{delcell}. 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:

\[ \text{ident1 [, ident2 ...]: [dynamic] array(list_of_sets) of type_name} \]

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. \text{set of real can be replaced by real in that context}).

\begin{verbatim}
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
\end{verbatim}
An array is of fixed size if all of its indexing sets are of fixed size (i.e. they are either constant or finalized (cf. procedure finalize)). If the qualifier dynamic is used, the array is dynamic and created empty. Otherwise (at least one indexing set is not constant), the array is created with as many cells as possible (i.e., the array is empty if one of the indexing sets is not initialized) and may grow if necessary. Such an array is not the same as a dynamic array even if it is created empty: Mosel may use a dedicated internal representation through which the creation of a single cell (via an assignment for instance) may induce the creation of a row of adjacent cells. The following example shows the different behaviour of an array that is simply declared with unknown index set (a) and an explicit dynamic array (b):

```mosel
declarations
  r: range
  a: array(r) of integer  ! a is created empty
  b: dynamic array(r) of integer  ! b is created empty
end-declarations
r:=1..3
finalize(r)  ! now the index set is known and constant
a(2):=1  ! here entries a(1) and a(3) are also created
b(2):=1  ! b(2) is the only entry of b
```

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.

### 2.6.4.1 Special case of dynamic arrays of a type not supporting assignment

Certain types do not have assignment operators: for instance, writing `x:=1` is a syntax error if `x` is of type `mpvar`. If an array of such a type 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 cells are not created. In that case, it is required to create each of the relevant entries of the array by using the procedure `create` since entries cannot be defined by assignment.

### 2.6.5 Records

A record is a finite collection of objects of any type. Each component of a record is called a field and is characterized by its name (an identifier) and its type. The general form of a record declaration is:

```mosel
ident1 [, ident2 ...]: record
  field1 [, field2 ...]: type_name
  [...] end-record
```

where `field_i` are the identifiers of the fields of the record and `type_name` one of the elementary types. Each of the identifiers `ident_i` is then declared as a record including the listed fields.

**Example:**

```mosel
declarations
  rl: record
    i,j:integer
    r:real
  end-record
end-declarations
```

Each record declaration is considered unique by the compiler. In the following example, although `rl1` and `rl2` have the same definitions, they are not of the same type (but `rl3` is of course of the type of `rl2`):

```mosel
declarations
  rl1: record
  end-record
```
2.6.6 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:

\[ \text{identifier} = \text{Expression} \]

where \text{identifier} is the name of the constant and \text{Expression} its initial and only value. The expression must be of one of the basic types, a set or a list of one of these types.

Example:

declarations
   STR='my const string'
   I1=12
   R=1..10 ! constant range
   S={2.3,5.6,7.01} ! constant set
   L=[2,4,6] ! constant list
end-declarations

The compiler supports two kinds of constants: a \textit{compile time constant} is a constant which value can be computed by the compiler. A \textit{run time constant} will be known only when the model is run.

Example:

parameters
   P=0
end-parameters
declarations
   I=1/3 ! compile time constant
   J=P*2 ! run time constant
end-declarations

2.6.7 User defined types

2.6.7.1 Naming new types

A new type may be defined by associating an identifier to a type declaration. The general form of a type definition is:

\[ \text{identifier} = \text{Type\_def} \]

where \text{Type\_def} is a type (elementary, set, list, array or record) to be associated to the symbol \text{identifier}. After such a definition, the new type may be used wherever a type name is required.

Example:

declarations
   entier=integer
   setint=set of entier
   i:entier ! <=> i:integer
   s:setint ! <=> s:set of integer
end-declarations

Note that only compile time constant or globally defined sets are allowed as indices to array types:
2.6.7.2  Combining types

Thanks to user defined types one can create complex data structures by combining structures offered by the language. For instance an array of sets may be defined as follows:

```mosel
declarations
  typset=set of integer
  a1:array(1..10) of typset
end-declarations
```

In order to simplify the description of complex data structures, the Mosel compiler can generate automatically the intermediate user types. Using this property, the example above can be written as follows (both arrays `a1` and `a2` are of the same type):

```mosel
declarations
  a2:array(1..10) of set of integer
end-declarations
```

2.7  Expressions

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

2.7.1  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, a list, an array or a record, 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:

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

where `array_ident` is the name of the array and `Exp_i` 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 ...]’`. In order to access the cell of an array of arrays, the list of indices for the second array has to be appended to the list of indices of the first array. For instance, the array `a:array(1..10) of array(1..10) of integer` can be dereferenced with `a(1,2)`.

Similarly, to access the field of a record, it is required to ‘dereference’ this record. The dereferencing of a record is denoted as follows:

```mosel
record_ident.field_ident
```

where `record_ident` is the name of the record and `field_ident` the name of the required field. Dereferencing arrays of records is achieved by combining the syntax for the two structures. For instance `a(1).b`

A function call is denoted as follows:
where \textit{function\_ident} is the name of the function and \textit{Expi} the \textit{i}th parameter required by this function. The first form is for a function requiring no parameter.

The special function \texttt{if} allows one to make a selection among expressions. Its syntax is the following:

\begin{verbatim}
  if (Bool\_expr, Exp1, Exp2)
\end{verbatim}

which evaluates to \textit{Exp1} if \textit{Bool\_expr} is \texttt{true} or \textit{Exp2} otherwise. The type of this expression is the type of \textit{Exp1} and \textit{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:
  \begin{verbatim}
  integer \rightarrow \texttt{real};
  \end{verbatim}

- in a function or procedure parameter list:
  \begin{verbatim}
  integer \rightarrow \texttt{real, lincr};
  real \rightarrow \texttt{lincr};
  mpvar \rightarrow \texttt{lincr};
  \end{verbatim}

- anywhere else:
  \begin{verbatim}
  integer \rightarrow \texttt{real, string, lincr};
  real \rightarrow \texttt{string, lincr};
  mpvar \rightarrow \texttt{lincr};
  boolean \rightarrow \texttt{string}.
  \end{verbatim}

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

\begin{verbatim}
! 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)
\end{verbatim}

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

\subsection*{2.7.2 Aggregate operators}

An operator is said to be \textit{aggregate} when it is associated to a list of indices for each of which a set or list of values is defined. This operator is then applied to its operands for each possible tuple of values (\textit{e.g.} the summation operator \texttt{sum} is an aggregate operator). The general form of an aggregate operator is:

\begin{verbatim}
Aggregate\_ident (Iterator1 [, Iterator2 ...]) Expression
\end{verbatim}

where the \texttt{Aggregate\_ident} is the name of the operator and \textit{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 \textit{Expression}.
An iterator is one of the following constructs:

\[
\begin{align*}
\text{SetList_expr} \\
\text{or} \\
\text{ident}_1 [, \text{ident}_2 ...] \text{ in SetList_expr} [\ | \ \text{Bool_expr}] \\
\text{or} \\
\text{ident} = \text{Expression} [\ | \ \text{Bool_expr}]
\end{align*}
\]

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 or list defined by \text{SetList_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 before the aggregate operator itself and not for each possible tuple of indices.

The Mosel compiler performs loop optimization when function \text{exists} is used as the first factors of the condition in order to enumerate only those tuples of indices that correspond to actual cells in the array instead of all possible tuples. To be effective, this optimization requires that sets used to declare the array on which the exist condition applies must be named and the same sets must be used to define the index domains. Moreover, the maximum speedup is obtained when order of indices is respected and all indices are defined in the same aggregate operator.

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).

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

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

2.7.4 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:
There are two basic operators for strings: the concatenation, written ‘+’ and the difference, written ‘−’.

\[ "a1b2c3d5"+"e6" \neq "a1b2c3d5e6" \]
\[ 'a1b2c3d5'-'3d5' \neq 'a1b2c' \]

### 2.7.5 Set expressions

Constant sets are described using one of the following constructs:

\[
\{ \text{Exp1} , \text{Exp2} \ldots \} \\
\text{integer\_exp1 . . integer\_exp2}
\]

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 intersec-
tion written ‘∗’.

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

\[
\{1,2,3\}+\{4,5,6\}-(5..8)\times(6,10) \neq \{1,2,3,4,5\} \\
\{'a','b','c'\}+\{'b','c','d'\} \neq \{'b','c'\} \\
\text{union}(i \text{ in } 1..4|i\neq2) \{i*3\} \neq \{3,9,12\}
\]

If several range sets are combined in the same expression, the result is either a range or a set of integers depending on the continuity of the produced domain. If range sets and sets of integers of more than one element are combined in an expression, the result is a set of integers. It is however possible to convert a set of integers to a range by using the notation range(setexpr) where setexpr is a set expression which result is either a set of integers or a range.

### 2.7.6 List expressions

A constant list consist in a list of expressions enclosed in square brackets:

\[
[\text{Exp1} , \text{Exp2} \ldots ]
\]

There are two basic operators for lists: the concatenation, written ‘+’ and the difference, written ‘−’. The aggregate operator sum can also be used to build up list expressions.

\[
[1,2,3]+[1,2,3] \neq [1,2,3,1,2,3] \\
[1,2,3,4]-[3,4] \neq [1,2] \\
\text{sum}(i \text{ in } 1..3) \{i*3\} \neq [3,6,9]
\]

### 2.7.7 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: \(<, \leq, =, \neq\) (not equal), \(\geq, >\).

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 <lower case letters <accented letters).

With sets, the comparators \(<\) (‘is subset of’), \(\geq\) (‘is superset of’), \(\neq\) (‘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.

With lists, the comparators = (‘equality of contents’) and <> (‘difference of contents’) are defined. These comparators must be used with two lists of the same type.

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.

\[ 3 \leq x \text{ and } y > 45 \text{ or } t < r \text{ and not } r \text{ in } \{1..10\} \]
\[ \text{and}(i \text{ in } 1..10) \ 3 \leq x(i) \]

### 2.7.8 Linear constraint expressions

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

The different forms of constraints are:

- \texttt{Linear_expr}
- \texttt{Linear_expr1 Ctr_cmp Linear_expr2}
- \texttt{Linear_expr SOS_type}
- \texttt{mpvar_ref mpvar_type1}
- \texttt{mpvar_ref mpvar_type2 Arith_expr}

In the case of the first form, the constraint is \textit{unconstrained} and is just a linear expression. For the second form, the valid comparators are \texttt{<=, >=, =}. The third form is used to declare special ordered sets. The types are then \texttt{is_sos1} and \texttt{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 \texttt{makesos1} or \texttt{makesos2} has to be used instead).

The last two types are used to set up special types for decision variables. The third series does not require any extra information: \texttt{is_continuous, is_integer, is_binary, is_free}. The second series of types is associated with a threshold value stated by an arithmetic expressions: \texttt{is_partint} for partial integer, the value indicates the limit up to which the variable must be integer, above which it is continuous. For \texttt{is_semcont} (semi-continuous) and \texttt{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.

\[ 3y + \text{sum}(i \text{ in } 1..10) \ x(i) \ i \geq z-t \]
\[ t \text{ is_integer} \]
\[ t \geq 7 \]
\[ \text{sum}(i \text{ in } 1..10) \ i \times x(i) \text{ is sos1} \]
\[ y \text{ is partint 5} \]
\[ y \leq 20 \]

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 \[ 3x+y=5 + y-10 \] is internally represented by: \[ 3x-5+y+10 \] 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 \texttt{ctl} refers to the linear constraint \[ 3x+5y-10 \], the expression \[ z-x+ctl \] is equal to: \[ z-2x-5y+10 \].

Note that the value of a unary constraint of the type \texttt{x is_type threshold is x-threshold}. 

\[ 3\times x \geq 5 \times y - 10 \]
\[ t \text{ is_integer} \]
\[ t \geq 7 \]
\[ \text{sum}(i \text{ in } 1..10) \ i \times x(i) \text{ is sos1} \]
\[ y \text{ is partint 5} \]
\[ y \leq 20 \]

Note that the value of a unary constraint of the type \texttt{x is_type threshold is x-threshold}. 

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 \[ 3x+y=5 + y-10 \] is internally represented by: \[ 3x-5+y+10 \] 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 \texttt{ctl} refers to the linear constraint \[ 3x+5y-10 \], the expression \[ z-x+ctl \] is equal to: \[ z-2x-5y+10 \].

Note that the value of a unary constraint of the type \texttt{x is_type threshold is x-threshold}. 

\[ 3\times x \geq 5 \times y - 10 \]
\[ t \text{ is_integer} \]
\[ t \geq 7 \]
\[ \text{sum}(i \text{ in } 1..10) \ i \times x(i) \text{ is sos1} \]
\[ y \text{ is partint 5} \]
\[ y \leq 20 \]
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 ';'.

2.8.1 Simple statements

2.8.1.1 Assignment

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

\[
\text{ident\_ref} := \text{Expression} \\
\text{or} \\
\text{ident\_ref} += \text{Expression} \\
\text{or} \\
\text{ident\_ref} -= \text{Expression}
\]

where \text{ident\_ref} is a reference to a value (i.e. an identifier or an array/record dereference) and \text{Expression} is an expression of a compatible type with \text{ident\_ref}. The \text{direct assignment}, denoted \(:=\) replaces the value associated with \text{ident\_ref} by the value of the expression. The \text{additive assignment}, denoted \(+\), and the \text{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.

\[
c:= 3*x+y \geq 5 \\
c+= y \quad ! \text{Implies } c \text{ is } 3*x+2*y-5 \geq 0 \\
c:= 3*x+y \geq 5 \\
c+: c + y \quad ! \text{Implies } c \text{ is } 3*x+2*y-5 \text{ (c becomes unconstrained)}
\]

2.8.1.2 Assignment of structured types

The direct assignment \(:=\) can also be used with sets, lists, arrays and records under certain conditions. For sets and lists, reference and value must be of the same type, the system performing no conversion on structures. For instance it is not possible to assign a set of integers to a set of reals although assigning an integer value to a real object is valid.

When assigning records, reference and value must be of the same type and this type must be \text{assignment compatible}; two records having identical definitions are not considered to be the same type by the compiler. In most cases it will be necessary to employ a user type to declare the objects. A record is assignment compatible if all the fields it includes can be assigned a value. For instance a record including a decision variable (type \text{mpvar}) cannot be used in an assignment: copying a value of such a type has to be performed one field at a time skipping those fields that cannot be assigned.

Two arrays can be used in an assignment if they have strictly the same definition and are
assignment compatible (i.e. their type supports assignment). Note that in a few cases arrays sharing the same definition cannot be assigned because their internal representations differ like in the following example:

```
declarations
 a:array(R:range) of integer ! 'a' is dynamic
end-declarations
R:=1..10
finalise(R)
declarations
 b:array(R) of integer ! 'b' is static
end-declarations
a:=b ! fails at run time
```

### 2.8.1.3 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.

```
! Assuming A,S,SE are unknown symbols
A:= 1 ! A is automatically defined
    ! as an integer reference
S:={1,2,3} ! S is automatically defined
    ! as a set of integers
SE:={} ! This produces a parser error as
    ! the type of SE is unknown
```

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.

```
A(1):=1 ! Implies: A:array(1..1) of integer
A(t):=2.5 ! Assuming "t in 1..10|f(t) > 0"
    ! implies: A:dynamic array(range) of real
```

The option `noimplicit` disables implicit declarations.

### 2.8.1.4 Inline initialization

Using *inline initialization* it is possible to assign several cells of an array in a single statement. The general form of an inline initialization is:

```
ident_ref :: [ Exp1 [, Exp2 ...] ]
```

or

```
ident_ref :: [Ind1 [, Ind2 ...] ][ Exp1 [, Exp2 ...] ]
```

where `ident_ref` is the object to initialize (array, set or list) and `Exp` are expressions of a compatible type with `ident_ref`. The first form of this statement may be used with lists, sets and arrays indexed by ranges: the list of expressions is used to initialize the object. In the case of lists and sets this operation is similar to a direct assignment, with an array, the first index of each dimension is the lower bound of the indexing range or 1 if the range is empty.

The second form is used to initialize regions of arrays or arrays indexed by general sets: each `Indi` expression indicates the index or list of indices for the corresponding dimension. An index list can be a constant, a list of constants (e.g. `[‘a’,‘b’,‘c’]`) or a constant range (e.g. `1..10`) but all values must be known at compile time.

```
declarations
 T:array(1..10) of integer
 U:array(1..9,{'a’,'b’,'c’}) of integer
end-declarations
```
2.8.1.5 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.

\[
10 \leq x; \quad x \leq 20
\]

\[
x \text{ is integer}
\]

2.8.1.6 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 or packages (cf. Section 2.3).

The general form of a procedure call is:

\[
\text{procedure_ident}
\]

\[
\text{procedure_ident}(\text{Exp1 [, \text{Exp2} ...])}
\]

where \text{procedure_ident} is the name of the procedure and, if required, \text{Exp}_i is the \text{i}th parameter for the call. Refer to Chapter 3 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.

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

2.8.2 Initialization block

The initialization block may be used to initialize objects (scalars, arrays, lists or sets) of basic type from files or to save the values of such objects to files. Scalars and arrays of external/user 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:

\[
\text{initializations from Filen}\mathit{ame}
\]

\[
\text{item}_1 [\text{as Label}_1]
\]

or

\[
[\text{item}_T11, \text{item}_T12 [, \text{Ident}_T13 ...]] \text{as Label}_T1
\]

\[
\text{item}_2 [\text{as Label}_2]
\]

or

\[
[\text{item}_T21, \text{item}_T22 [, \text{Ident}_T23 ...]] \text{as Label}_T2
\]

\[
\text{end-initializations}
\]

where Filen\mathit{ame}, a string expression, is the name of the file to read, item\_i any object identifier and item\_Tij 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. If a given item is of a record type, the operation is permitted only if all fields it contains can be initialized. For instance, if one of the fields is a decision variable (type mpvar), the compilation will fail. Alternatively, the fields to be initialized can be listed using the following syntax as an item:
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. Moreover, if control parameter readcnt is set to true before the execution of the block, counting is also achieved at the label level: the number of items actually read in for each label may be obtained using function getreadcnt.

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 lists, sets records 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 (FALSE) and true (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:

```
initializations to Filename
item1 [as Label1]
or
itemT11, itemT12 [ , IdentT13 ...]asLabelT1
|
item2 [ as Label2]
or
itemT21, itemT22 [ , IdentT23 ...]asLabelT2
...
end-initializations
```

In this form, any itemi can be replaced the value of an expression using the following construct (Labeli is mandatory in this case):

```
evaluation of expression
```

When this second form is executed, the value of all provided labels is updated with the current value of the corresponding identifier\(^1\) 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:

\(^{1}\)A copy of the original file is saved prior to the update (i.e. the original version of fname can be found in fname˜).
consider the following program:

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

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:

```mosel
! Example of the use of initialization blocks

! 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
'
```

In case of error (e.g. file not found, corrupted data format) during the processing of an initialization block, the execution of the model is interrupted. However if the value of control parameter ioctrl is true, executions continues. It is up to the user to verify whether data has been properly transfered by checking the value of control parameter iostatus.

### 2.8.3 Selections

#### 2.8.3.1 If statement

The general form of the `if` statement is:

```mosel
if Bool_exp_1
  then Statement_list_1
  [else Bool_exp_2
    then Statement_list_2
    [...]]
  [else Statement_list_E]
end-if
```

The selection is executed as follows: if `Bool_exp_1` is `true` then `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 `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
```

### 2.8.3.2 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: `Expression_0` is evaluated and compared sequentially with each expression of the list `Expression_i` until a match is found. Then the statement `Statement_i` (resp. list of statements `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 `Statement_list_E` is executed, otherwise the execution continues after the `end-case` instruction. Note that, each of the expression lists `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.

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

### 2.8.4 Loops

#### 2.8.4.1 Forall loop

The general form of the `forall` statement is:

```plaintext
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 (cf. Section 2.7.2).

```plaintext
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
```
2.8.4.2 While loop

The general form of the while statement is:

\[
\text{while} \ (\text{Bool} \_\text{expr}) \ \text{Statement} \\
or \\
\text{while} \ (\text{Bool} \_\text{expr}) \ \text{do} \ \text{Statement} \_\text{list} \ \text{end-do}
\]

The statement \text{Statement} (resp. list of statements \text{Statement} \_\text{list}) is repeated as long as the condition \text{Bool} \_\text{expr} is true. If the condition is false at the first evaluation, the while statement is entirely skipped.

\[
i:=1 \\
\text{while}(i<=10) \ \text{do} \ \\
\quad \text{write}(\ ' ',i) \\
\quad \text{if isodd}(i) \ \text{then} \ s+={i} \\
\quad \text{end-if} \\
\quad i+=1 \\
\text{end-do}
\]

2.8.4.3 Repeat loop

The general form of the repeat statement is:

\[
\text{repeat} \ \\
\quad \text{Statement1} \\
\quad / \ \text{Statement2} \ ... / \\
\quad \text{until} \ \text{Bool} \_\text{expr}
\]

The list of statements enclosed in the instructions repeat and until is repeated until the condition \text{Bool} \_\text{expr} is true. As opposed to the while loop, the statement(s) is (are) executed at least once.

\[
i:=1 \ \\
\text{repeat} \ \\
\quad \text{write}(\ ' ',i) \\
\quad \text{if isodd}(i) \ \text{then} \ s+={i} \\
\quad \text{end-if} \\
\quad i+=1 \\
\text{until} \ i>10
\]

2.8.4.4 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:

\[
\text{break} \ [n] \\
or \\
\text{next} \ [n]
\]

where \( n \) is an optional integer constant: \( n-1 \) nested loops are stopped before applying the operation.
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)
brack ! 9: Stop L2 and continue after line 10
dend-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.

2.9.1 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 (cf. Section 2.6) 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_parms]): Type
  Func_body
end-function
```

The only difference with a procedure is that the function type must be specified: it can be any type name except `mpvar`. 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

2.9.2 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

2.9.3 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 referenced outside of the routine (e.g. member of a set defined globally). As a consequence, active constraints (linctr that are not just linear expressions) declared inside a subroutine and the variables they employ are still effective after the termination of the subroutine (because they are part of the current problem) even if the symbols used to name the related objects are 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)

This code extract displays:

Inside of myproc, i=a string j=4
Outside of myproc, i=45 j=4

2.9.4 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.

2.9.5 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:

\[
\text{forward procedure Proc\_name \left[\text{List\_of\_params}\right]} \\
\text{or} \\
\text{forward function Func\_name \left[\text{List\_of\_params}\right]: Basic\_type}
\]

where the procedure or function Func\_name will be defined later in the source file. Note that a forward declaration for which no actual definition can be found is considered as an error by Mosel.

forward function f2(x:integer):integer

function f1(x:integer):integer
  returned:=x+if(x>0,f2(x-1),0) ! f1 needs to know f2
end-function

function f2(x:integer):integer
  returned:=x+if(x>0,f1(x-1),0) ! f2 needs to know f1
end-function

2.9.6 Suffix notation

Functions which name begins with get and taking a single argument may be called using a suffix notation. This alternative syntax is constructed by appending to the variable name (the intended function parameter) a dot followed by the function name without its prefix get. For instance the call getsol(x) is the same as x.sol. The compiler performs internally the translation from the suffix notation to the usual function call notation, the two syntaxes are equivalent.

Similarly, calls to procedures which name begins with set and taking two arguments may be written as an assignment combined with a suffix notation. In this case the statement can be replaced by the variable name (the intended first procedure parameter) followed by a dot and the procedure name without its prefix set then the assignment sign := and the value corresponding to the second parameter. For instance the statement sethidden(ctl, true) can also be written ctl.hidden:=true. As for the other alternative notation, the compiler performs the rewriting internally and the two syntaxes are equivalent.
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 `print` 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 (except parameters) are considered to be private.

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

```
public declarations
   e:integer ! e is published
   f:integer ! f is published
end-declarations

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
```

This qualifier can also be used when declaring record types in order to select the fields of the record that can be accessed from outside of the file making the definitions: this allows to make available only a few fields of a record, hiding what is considered to be internal data.

```
declarations
   public t1=record
      i:integer ! t1.i is private
      public j:real ! t1.j is public
   end-record
   public t2=public record
      i:integer ! t2.i is public
      j:real ! t2.j is public
   end-record
end-declarations
```

2.11 Packages

Declarations may be stored in a package: once compiled, the package can be used by any model by means of the `uses` statement. Except for its beginning and termination (keyword `model` is replaced by `package`) a package source is similar to a normal model source. The following points should be noticed:

- all statements and declarations outside procedure or function definitions are used as an initialization routine: they are automatically executed before statements of the model using the package;
- symbols that should be published by the package must be made explicitly public using the `public` qualifier (see Section 2.10);
- parameters of a package are automatically added to the list of parameters of the model using the package;
- as opposed to modules that are dynamically linked, bim files of packages are used only at compilation time — they are not required for execution;
- a package cannot be imported several times by a model and packages publish symbols of packages they use. For instance, assuming package P1 imports package P2, a model using
P1 cannot import explicitly P2 (with a uses statement) but has access to the functionality of P2 via P1.

2.11.1 The requirements block

Requirements are symbols a package requires for its processing but does not define. These required symbols are declared in requirement blocks which are a special kind of declaration blocks in which constants are not allowed but procedure/functions can be declared. The symbols of such a block have to be defined when the model using the package is compiled: the definitions may appear either in the model or in another package but cannot come from a module. Several packages used by a given model may have the same requirements (i.e. same identifier and same declaration). It is also worth noting that a package inherits the requirements of the packages it uses.

requirements
   an_int:integer
   s0: set of string
   bigar: array(S0) of real
   procedure doit(i:integer)
end-requirements

2.12 File names and input/output drivers

Mosel handles data streams using IO drivers: a driver is an interface between Mosel and a physical data source. Its role is to expose the data source in a standard way such that from the user perspective, all data sources can be accessed using the same methods (i.e. initializations blocks, file handling functions). Drivers are specified in file names: all Mosel functions supporting IO operations though drivers can be given an extended file name. This type of name is composed of the pair driver_name:file_name. When Mosel needs to access a file, it looks for the specified driver in the table of available drivers. This table contains all predefined drivers as well as drivers published by modules currently loaded in memory. If the driver is provided by a module, the module name may also be indicated in the extended file name: module_name.driver_name:file_name. Using this notation, Mosel loads the required module if necessary (otherwise the file operation fails if the module is not already loaded). For instance it is better to use mmodbc.odbc:database than odbc:database.

The file_name part of the extended file name is specific to the driver and its structure and meaning depends on the driver. For instance, the sysfd driver expects a numerical file descriptor so file sysfd:1 is a valid name but sysfd:myfile cannot work. A driver may act as a filter and expects as file_name another extended file name (e.g. zlib.compress:mem:0x123/23).

When no driver name is specified, Mosel uses the default driver which name is an empty string (myfile is equivalent to :myfile). This driver relies on OS functions to access files from the file system. The null driver can be used to disable a stream: whatever written to file "null:" is ignored and reading from it is like reading from an empty file.

The tee driver can only be open for writing and expects as file name a list of up to 6 extended file names separated with '&': it opens all the specified files and duplicates what it receives to each of them. If only one file is given or if the string terminates with '&', output is also sent to the default output stream (or error stream if the file is used for errors). For instance, writing to the file "tee:log1&log2&" has the effect of writing at the same time to files "log1" and "log2" as well as sending a copy to the console.

The other predefined drivers (sysfd, mem, cb and raw) are useful when interfacing Mosel with a host application. They are described in detail in the Mosel Libraries Reference Manual.

2.13 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, fflush). 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.

```mosel
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")
end-model
```

A copy of the original file is saved prior to the update (i.e. the original version of fname can be found in fname˜).
Chapter 3
Predefined functions and procedures

This chapter lists in alphabetical order all 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 $\pi$
**Purpose**
Get the absolute value of an integer or real.

**Synopsis**

```plaintext
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 an integer or real number.

**Further information**
This function returns the absolute value of an integer or real number. The returned type corresponds to the type of the input.

**Related topics**
exp, ln, log, sqrt.
arctan

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

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

**Argument**
`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:
```plaintext
function arcsin(s:real):real
  returned:=arctan(s/sqrt(1-s^2))
end-function

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

**Related topics**
`cos, sin`
assert

Purpose
Abort execution if a condition is not satisfied.

Synopsis
procedure assert(c:boolean)
procedure assert(c:boolean,m:string)
procedure assert(c:boolean,m:string,e:integer)

Arguments
  c  Condition to verify
  m  Error message to display in case of failure
  e  Error code to return in case of failure (default: 8)

Example
assert(and(i in I) mydata(i)>0)
assert(isodd(a),"a is not odd!!")

Further information
1. If the condition \( c \) is satisfied, this procedure has no effect, otherwise it displays an error message and aborts execution by calling \texttt{exit}. The versions of the procedure with 2 and 3 parameters can be used to replace the default message (location of the statement in the source) and default exit value (8).

2. Assertions are usually used as a debugging tool and are ignored when the model is compiled without debugging information (\textit{i.e.} none of options \texttt{-g} or \texttt{-G} is used). It is however possible to keep assert statements even when no debugging information is included by specifying the compiler directive \texttt{keepassert} (see Section 2.3).

Related topics
exit
bittest

Purpose
Test bit settings.

Synopsis
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 takes the value 5, and k takes the value 8:

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

**Synopsis**

```plaintext
function ceil(r:real):integer
```

**Argument**
r  
Real number to be rounded

**Return value**
Rounded value.

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

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

**Related topics**
floor, round.
**Purpose**  
Get the cosine of a value.

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

**Argument**  
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:

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

**Related topics**  
arctan, sin.
create

Purpose
Create explicitly a cell of a dynamic array.

Synopsis
procedure create(x:array reference)

Argument
x     Cell 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 $x(1) + x(3) + x(5) + x(7)$:

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

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

Further information
Usually cells of dynamic arrays are created by means of assignments. This procedure can be used as a replacement for an assignment especially when the type of a dynamic array does not provide any assignment operator (like mpvar for instance).

Related topics
Section 2.6.4, delcell.
Purpose
Check if a given entry in a dynamic array has been created.

Synopsis
function exists(x):boolean

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

Return value
true if the entry exists, false otherwise.

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

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

forall(i in S| not isodd(i)) create(x(i))
forall(i in S| exists(x(i)))
writeln("x(" , i , ") exists")

Further information
1. 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.

2. Under certain conditions, the exists function call is optimized by the compiler when used for filtering an aggregate operator: the loop is only performed for the existing entries instead of enumerating all possible tuples of indices for finding the relevant ones.

Related topics
Section 2.7.2, create, create.
Purpose
Terminate the program.

Synopsis
procedure exit(code:integer)

Argument
code   Value to be returned by the program

Further information
This procedure terminates the current program and returns the given value. Models exit by default with a value of 0 unless this is changed using exit. The Mosel command line interpreter uses this value as exit status.

Related topics
Section 1.3.
exp

Purpose
Get the natural exponent of a value.

Synopsis
function exp(r: real): real

Argument
r Real value the function is applied to.

Return value
Natural exponent (e^r) of the argument.

Related topics
abs, exp, ln, log, sqrt.
Exportprob

Purpose
Export a problem to a file.

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

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

Example
The following prints the current problem to the screen using the default format and with MinCost as objective function. The second statement exports the problem in LP-format and with scrambled names to the file prob1.lp maximizing the constraint Profit:

declarations
  MinCost, Profit:linctr
end-declarations

exportprob(0, "", MinCost)
exportprob(EP_MAX+EP_STRIP, "prob1", Profit)

Further information
1. If the given filename uses the default IO driver (no driver specified) and has no extension, Mosel appends .lp to it for LP format files and .mat for MPS format.
2. Normally, local symbols (i.e. defined in a procedure or function) are replaced by generated names in the exported matrix. However, if the model has been compiled with option -G, names defined locally to the routine calling exportprob are used in the exported matrix. Moreover, if a local symbol hides a global one, this symbol is prefixed by "˜".
3. If no option is provided, the default format is LP for a minimization; if no constraint is given, the current objective (if available) is exported. The matrix is printed to the standard output when this function is used without parameter.
**currentdate**

**Purpose**
Return the current date as a Julian Day Number (JDN).

**Synopsis**
```
function currentdate:integer
```

**Return value**
The number of days elapsed since 1/1/1970 as an integer.

**Further information**
1. The control parameter "UTC" indicates whether this function returns a date in local or UTC time.
2. Refer to the module `mmsystem` for a set of dedicated types for handling date and time.

**Related topics**
`setparam`, `timestamp`, `currenttime`
currenttime

**Purpose**
Return the current time as the number of milliseconds since midnight.

**Synopsis**
```plaintext
function currenttime:integer
```

**Return value**
The number of milliseconds since midnight as an integer.

**Further information**
1. The control parameter "UTC" indicates whether this function returns a time in local or UTC time.
2. Refer to the module `mmsystem` for a set of dedicated types for handling date and time.

**Related topics**
- `setparam`
- `timestamp`
- `currentdate`
**cuthead**

**Purpose**
Cut the first elements of a list.

**Synopsis**

```
procedure cuthead(l:list, o:integer)
```

**Arguments**
- `l` A list
- `o` Number of elements to remove if >0 or number of elements to keep if <0

**Example**
```
L:=[1,2,3,4,5]
cuthead(L,2) ! => L=[3,4,5]
cuthead(L,-1) ! => L=[5]
```

**Further information**
If the second parameter is 0, the list is unchanged. If the same parameter is larger than the size of the list, all elements are deleted.

**Related topics**
- `cuttail`
cuttail

**Purpose**
Cut the last elements of a list.

**Synopsis**
procedure cuttail(l:list, o:integer)

**Arguments**
- **l** A list
- **o** Number of elements to remove if >0 or number of elements to keep if <0

**Example**
```
L:=[1,2,3,4,5]
cuttail(L,2) ! => L=[1,2,3]
cuttail(L,-1) ! => L=[1]
```

**Further information**
If the second parameter is 0, the list is unchanged. If the same parameter is larger than the size of the list, all elements are deleted.

**Related topics**
cuthead
delcell

**Purpose**
Delete a cell or all cells of a dynamic array.

**Synopsis**

```plaintext
procedure delcell(x:array reference)
procedure delcell(a:array)
```

**Arguments**
- `x` Cell to be deleted
- `a` An array

**Further information**
Only cells of arrays explicitly declared dynamic can be deleted. This function has no effect with other types of array.

**Related topics**
Section 2.6.4, `create`. 
fclose

Purpose
Close the active input or output stream.

Synopsis
procedure fclose(stream:integer)

Argument
stream  The stream to close:

F_INPUT  Input stream
F_OUTPUT  Output stream

Further information
This procedure 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
fflush, fopen, fselect, getfid, iseof.
**fflush**

**Purpose**
Force the operating system to write 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**
`fclose`, `fopen`.

Predefined functions and procedures

Mosel Reference Manual
finalize

Purpose
Finalize the definition of a set or list.

Synopsis
procedure finalize(s:set)
procedure finalize(l:list)

Arguments
s    Dynamic set
l    Dynamic list

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:

```mosel
declarations
Set1: set of string
  x: array(Set1) of mpvar ! Declare a dynamic array of variables
    ! (entries need to be created
    !     subsequently)
end-declarations
Set1:= ("first", "second", "fifth")
finalize(Set1) ! Finalize the set definition

declarations
  y: array(Set1) of mpvar ! Declare a static array of variables
    ! (entries are created immediately)
end-declarations
```

Further information
This procedure finalizes the definition of a set (or list), 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 (= 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.
findfirst

**Purpose**
Find the first occurrence of an element in a list.

**Synopsis**

```mosel
function findfirst(l:list, e:type_of_l):integer
```

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>l</td>
<td>A list</td>
</tr>
<tr>
<td>e</td>
<td>The element to look for (it must be of the type of l)</td>
</tr>
</tbody>
</table>

**Return value**
The position of the element or 0 if the element is not included in the list.

**Example**

```mosel
L:=['a','b','c','d','b']
i:=findfirst(L,'b') ! => i=2
i:=findlast(L,'f') ! => i=0
```

**Related topics**

*findlast*
findlast

Purpose
Find the last occurrence of an element in a list.

Synopsis
function findlast(l:list, e:type_of_l):integer

Arguments
- l A list
- e The element to look for (it must be of the type of l)

Return value
The position of the element or 0 if the element is not included in the list.

Example
L:=['a','b','c','d','b']
i:=findlast(L,'b') ! => i=5
i:=findlast(L,'f') ! => i=0

Related topics
findfirst
floor

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

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

**Argument**
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**
ceil, 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 (may be combined):

- 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
- F_LINBUF If open for writing, flushes buffer after end of each line (default when writing to a console)
- F_SILENT Do not display IO error messages

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 behavior 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 behavior 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**
procedure fselect(stream:integer)

**Argument**
- stream The stream number

**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.

```mosel
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 concerned stream 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)

**Argument**
filter List of comment signs

**Example**
In the following, the first statement skips all lines beginning with either '#', '!', or '. The second statement skips any following blank lines:

```plaintext
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

Argument
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 a constraint coefficient or constant term.

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)
r:= 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

Argument
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 and the constraint is contained in the problem, 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

Argument
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.
**getfname**

**Purpose**
Get the file name associated to the active input, output or error stream.

**Synopsis**
```
function getfname(stream:integer):string
```

**Argument**
- **stream** The stream to query:
  - F_INPUT Input stream
  - F_OUTPUT Output stream
  - F_ERROR Error stream

**Return value**
File name.
getfirst

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

**Synopsis**

```plaintext
function getfirst(r:range):integer
function getfirst(l:list):type_of_l
```

**Arguments**

- `r` A range set
- `l` A list

**Return value**

The first element of the set or list.

**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**

When applied to a list, the type of the function is the type of the list. An error is generated if the argument of the function is empty.

**Related topics**

- `getlast`
**gethead**

**Purpose**
Get a copy of the first elements of a list.

**Synopsis**
function gethead(l:list, o:integer):list

**Arguments**
- **l**: A list
- **o**: Number of elements to copy if >0 or number of elements to ignore if <0

**Return value**
A (partial) copy of the list.

**Example**
```
L:=[1,2,3,4,5]
L2:=gethead(L,2) ! => L2=[1,2]
L2:=gethead(L,-1) ! => L2=[1,2,3,4]
```

**Further information**
This function does not alter its input list. If the second parameter is 0 an empty list is returned. If the same parameter is larger than the size of the list the function returns a copy of the original list.

**Related topics**
- gettail
getlast

Purpose
Get the last element of a range set or a list.

Synopsis
function getlast(r:range):integer
function getlast(l:list):type_of_l

Arguments
r A range set
l A list

Return value
The last element of the set or list.

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

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

Further information
When applied to a list, the type of the function is the type of the list. An error is generated if the argument of the function is empty.

Related topics
getfirst.
**getobjval**

**Purpose**  
Get the objective function value.

**Synopsis**  
```plaintext  
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. 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

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

Return value
Current setting of the control parameter.

Further information
1. Parameters whose values may be returned by this function 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)
   - ioctrl   the interpreter ignores IO errors (Boolean)
   - iostatus status of the last IO operation (integer)
   - nbread   number of items recognized by the last read procedure or read in by the last initializations block (integer)
   - readcnt  generate per label counting when executing 'initializations from' blocks (Boolean)
   - UTC      indicate whether the time functions return time expressed in local (false) or UTC (true) time (Boolean)

4. Function getparam may also be used to retrieve parser parameters. As opposed to the other parameters whose value is computed at run time, these parameters are evaluated as soon as they are parsed:
   - parser_line number of the line being parsed (integer)
   - parser_file current source file name (string)
   - parser_date current local date (string)
   - parser_time current local time (string)
   - parser_UCTdate current UTC date (string)
   - parser_UCTtime current UTC time (string)
   - parser_version Mosel version (string)
   - model_version Version of the model as given by the version directive (string)

Related topics
setparam.
getrcost

Purpose
Get the reduced cost value of a variable.

Synopsis
function getcost(v:mpvar):real

Argument
v           A decision variable

Return value
Reduced cost value or 0.

Further information
This function returns the reduced cost value of a variable if the problem has been solved successfully and the variable is contained in the problem, otherwise 0 is returned.

Related topics
getslack, getsol, getdual.
getreadcnt

**Purpose**
Get the number of items read in during last ‘initializations from’ for a given label.

**Synopsis**
function getreadcnt(l:string):integer

**Argument**
l
A label

**Return value**
Number of items read in for label l.

**Further information**
Value 0 is returned if the given string does not correspond to a label or if control parameter readcnt has not been set to true before execution of the initializations block.
getreverse

**Purpose**
Duplicate and reverse a list.

**Synopsis**
```
function getreverse(l:list):list
```

**Argument**
- `l` A list

**Return value**
A reversed copy of the provided list.

**Example**
```
L:=[1,2,3,4,5]
L2:=L.reverse ! => L=[5,4,3,2,1]
```

**Related topics**
reverse.
getsize

Purpose
Get the size of an array, set, list, constraint or string.

Synopsis
function getsize(a:array):integer
function getsize(s:set):integer
function getsize(l:list):integer
function getsize(t:string):integer
function getsize(c:linctr):integer

Arguments
a  An array
s  A set
l  A list
t  A string
c  A linear constraint

Return value
Number of effective entries for an array, number of elements for a set or a list, number of characters for a string, number of terms for a constraint.

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. The length lw of the string w is 9.

declarations
  a:dynamic array(1..8) of real
  w = "some text"
end-declarations

  a(1):= 4
  a(5):= 7.2
  la:= getsize(a)
  lw:= getsize(w)

Further information
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 string (i.e. the number of characters it contains). If used with a linear constraint, this function returns the number of terms of the constraint (the constant term is not taken into account).
getslack

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

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

**Argument**
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 and the constraint is contained in the problem, otherwise 0 is returned.

**Related topics**
getdual, getrcost, getsol.
**getsol**

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

**Synopsis**

```plaintext
function getsol(v:mpvar):real
function getsol(c:linctr):real
```

**Arguments**

- `c` A linear constraint
- `v` A decision variable

**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`
- `getcost`
- `getobjval`
gettype

Purpose
Get the type of a constraint.

Synopsis
function gettype(c:linctr):integer

Argument
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    Non-binding 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
CT_FREE  Free

Related topics
settype.
gettail

**Purpose**
Get a copy of the last elements of a list.

**Synopsis**
```mosel
function gettail(l:list, o:integer):list
```

**Arguments**
- `l`: A list
- `o`: Number of elements to copy if >0 or number of elements to ignore if <0

**Return value**
A (partial) copy of the list.

**Example**
```mosel
L:=[1,2,3,4,5]
L2:=gettail(L,2) ! => L2=[4,5]
L2:=gettail(L,-1) ! => L2=[2,3,4,5]
```

**Further information**
This function does not alter its input list. If the second parameter is 0 an empty list is returned. If the same parameter is larger than the size of the list the function returns a copy of the original list.

**Related topics**
- `gethead`
getvars

Purpose
Get the set of variables of a constraint.

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

Arguments
- c  A linear constraint
- s  A set of decision variables

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:

  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:

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

**Further information**
This function returns the “end of file” status of the active input stream.

**Related topics**
fclose, fopen.
ishidden

Purpose
Test whether a constraint is hidden.

Synopsis
function ishidden(c:linctr):boolean

Argument
c A linear constraint

Return value
true if the constraint is hidden, false otherwise.

Further information
This function tests the current status of a constraint. At its creation a constraint is added to the current problem, but using the function sethidden it may be hidden. This means, 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**
```mosel
function isodd(i:integer):boolean
```

**Argument**
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
function ln(r:real):real

Argument
r          Real value the function is applied to. This value must be positive.

Return value
Natural logarithm of the argument.

Example
The following example provides a function for calculating logarithms to any (positive) base:

    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**
function log(r: real): real

**Argument**
r
Real value the function is applied to. This value must be positive.

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

**Related topics**
exp, ln, sqrt.
makesos1, makesos2

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

**Synopsis**

```plaintext
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.

```plaintext
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 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 \( \text{maxlist} \):

\[
r := \text{minlist}(-1, 4.5, 2, 7, -0.3)
\]

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 between 0 and 1.

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

```mosel
d i := integer(round((10*random)+0.5))
```

**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**

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

**Argument**
e1, e2,... Expression or list of expressions of basic type

**Example**
The following reads (possible split over several lines) 12 45 word, followed by `toto(12 and 45)=word`:

```plaintext
declarations
  i,j:integer
  s:string
  ts:array (range,range) of string
end-declarations
read(i, j, s)
readln("toto(" , i, "and" , j, ")=" , 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 `nbread` to the number of items actually recognized.

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 `readln` expects all the items to be recognized to be contained in single line. The function `read` ignores changes of line. If the procedure `readln` is used without parameters it skips the end of the current line.

**Related topics**

write, write.
reverse

Purpose
Reverse a list.

Synopsis
procedure reverse(l:list)

Argument
l       A list

Example
L:=[1,2,3,4,5]
reverse(L) ! => L=[5,4,3,2,1]
reverse(L) ! => L=[1,2,3,4,5]

Related topics
greverse.
round

**Purpose**
Round a number to the nearest integer.

**Synopsis**

function round(r:real):integer

**Argument**

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 13:

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

**Related topics**
ceil, floor.
setcoeff

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

**Synopsis**
```
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:
```
declarations
x,y,z: mpvar
end-declarations

\( c := 4 \times x + y - z \leq 12 \)

setcoeff(c, y, 2)
setcoeff(c, 8.1)
```
The constraint is now \( 4 \cdot x + 2 \cdot y - z \leq -8.1 \).

**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**
```plaintext
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:

```plaintext
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 a control parameter (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_verbose). The type of the value must correspond to the type expected by the
parameter.

2. This procedure can be applied only to control parameters the value of which 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 in comparisons between reals (real, default: 1.0e-13)
ioctrl the interpreter ignores IO errors (Boolean, default: false)
readcnt generate per label counting when executing ‘initializations from’ blocks (Boolean,
default: false)
UTC indicate whether the time functions return time expressed in local (false) or UTC (true)
time (Boolean, default: false)

Related topics
getparam.
setrandseed

Purpose
Initialize the random number generator.

Synopsis
procedure setrandseed(s:integer)

Argument
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  Non-binding 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
CT_FREE Free

Related topics
gettype
sin

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

**Synopsis**

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

**Argument**

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

**Return value**

Sine value of the argument.

**Related topics**

- `arctan`
- `cos`
splithead

Purpose
Split a list returning the first elements.

Synopsis
function splithead(l:list, o:integer):list

Arguments
l A list
o Number of elements to remove if >0 or number of elements to keep if <0

Return value
The list of elements removed.

Example
L:=[1,2,3,4,5]
L2:=splithead(L,2) ! => L=[3,4,5] L2=[1,2]

Further information
If the second parameter is 0, the list is unchanged and an empty list is returned. If the same parameter is larger than the size of the list, all elements are deleted and the function returns a copy of the original list.

Related topics
splittail
**Purpose**
Split a list returning the last elements.

**Synopsis**
function splittail(l:list, o:integer):list

**Arguments**
l A list
o Number of elements to remove if >0 or number of elements to keep if <0

**Return value**
The list of elements removed.

**Example**
L:=[1,2,3,4,5]
L2:=splittail(L,2) ! => L=[1,2,3] L2=[4,5]
L2:=splittail(L,-1) ! => L=[1] L2=[2,3]

**Further information**
If the second parameter is 0, the list is unchanged and an empty list is returned. If the same parameter is larger than the size of the list, all elements are deleted and the function returns a copy of the original list.

**Related topics**
splithead
sqrt

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

**Synopsis**

```plaintext
function sqrt(r:real):real
```

**Argument**

- `r` Real value the function is applied to. This value must be non-negative.

**Return value**

Square root of the argument.

**Related topics**

`abs, exp, ln, log`. 
strfmt

Purpose
Create 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 this 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))

produces this output:
    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**
```plaintext
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**
```plaintext
write(substr("Example text", 3, 10))
```

This outputs the text: ample te

**Further information**
This function returns the substring from the \(i_1^{th}\) to the \(i_2^{th}\) 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).
**timestamp**

**Purpose**
Generate a timestamp by combining the current UTC date and time.

**Synopsis**
function timestamp:real

**Return value**
The number of seconds since 1/1/1970 at midnight as a real.

**Further information**
1. This function corresponds to the expression (using UTC time):
   \[ \text{real}(\text{currentdate}) \times 86400 + \text{currenttime}/1000 \]
2. Refer to the module **mmsystem** for a set of dedicated types for handling date and time.

**Related topics**
currenttime, currentdate
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...])

Argument
e1, e2,... Expression or list of expressions

Example
The following lines

    Set1:={"first", "second", "fifth"}
    write(Set1) ! Print set contents without return
    writeln ! Print an empty line
    b:=true
    writeln("A real:", strfmt(7.1234, 4, 2), ", a Boolean:",b)
    ! Output followed by return

produce this output:

    {'first', 'second', 'fifth'}
    A real:7.12, a Boolean:true

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.
II. Modules
Chapter 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>Procedure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>disc</td>
<td>Annual discount.</td>
<td>110</td>
</tr>
<tr>
<td>diskdata</td>
<td>Read in or write an array or set of strings to a file.</td>
<td>111</td>
</tr>
</tbody>
</table>
disc

Purpose
Annual discount.

Synopsis
function disc(a:real, t:real)

Arguments
a  Discount factor, real number greater than -1
   t  Time, real number

Return value
Annual discount value: 1 / (1 + a)^t - 1.

Further information
This function calculates the annual discount for the given period of time and discount factor.
**diskdata**

**Purpose**
Read in or write an array or set of strings to a file.

**Synopsis**

procedure diskdata(format:integer, file:string, a:array)
procedure diskdata(format:integer, file:string, s:set of string)

**Arguments**

- **format**  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 a file
  - ETC_APPEND append output to the end of an existing file
  - ETC_TRANS tables are transposed
  - ETC_IN read from file (default)
  - ETC_NOZEROS skip zero values
  Several options may be combined using '+'.

- **file**  Extended file name

- **a**  Array with elements of a basic type

- **s**  Set of strings

**Example**
The following example declares two sets and two dynamic arrays. The array `ar1` is read in from the file `in.dat`. Then both arrays, `ar1` and `ar2`, are saved to the file `out.dat` (in sparse format) and finally the contents of the set `Set1` is appended to the file `out.dat`.

```mosel
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 in 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.
4.2 I/O drivers

This module provides the diskdata IO driver designed to be used as an interface for initializations blocks for both reading and writing files formatted for the diskdata procedure.

4.2.1 Driver diskdata

diskdata: [dense, sparse, sglq, noq, append, trans, nozeros]

The driver can only be used in ‘initializations’ blocks. In the opening part of the block, no file name has to be provided, but general options can be stated at this point: they will be applied to all labels. In the block, each label entry is understood as the file name to use for the actual processing. Note that, before the file name, one can add further options separated by commas, that are effective to the particular entry. The file name given can use extended notation.

The diskdata driver takes the following options:

dense dense data format
sparse sparse data format
sglq strings quoted with single quotes
noq  strings are not quoted in the file
append append output to the end of an existing file
trans tables are transposed
nozeros skip zero values

Example:

declarations
Set1: set of string ! Declare a set of strings
ar1, ar2: array(Set1, range) of real ! Declare two dynamic arrays
r: real ! Declare a real value
end-declarations

initializations from "diskdata:"
ar1 as "sparse, ind.dat" ! Read ‘ar1’ from ‘in.dat’ in sparse format
r as "r_init.dat" ! Initialize ‘r’ from ‘r_init.dat’
end-initializations

initializations to "diskdata:append"
[ar1, ar2] as "out.dat" ! Save two arrays in sparse format
Set1 as "out.dat" ! Save set ‘Set1’ to the same file
end-initializations
Chapter 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:

```plaintext
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>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>IVE_RGB</td>
<td>Compute a composite color.</td>
<td>114</td>
</tr>
<tr>
<td>IVEaddplot</td>
<td>Insert a new plot on the user graph.</td>
<td>115</td>
</tr>
<tr>
<td>IVEdrawarrow</td>
<td>Add an arrow to an existing plot.</td>
<td>116</td>
</tr>
<tr>
<td>IVEdrawellipse</td>
<td>Add a filled ellipse to an existing plot.</td>
<td>125</td>
</tr>
<tr>
<td>IVEdrawgantt</td>
<td>Draw a Gantt chart with CP operations.</td>
<td>117</td>
</tr>
<tr>
<td>IVEdrawlabel</td>
<td>Add a text box to an existing plot.</td>
<td>118</td>
</tr>
<tr>
<td>IVEdrawline</td>
<td>Add a line to an existing plot.</td>
<td>119</td>
</tr>
<tr>
<td>IVEdrawpoint</td>
<td>Add a small square to mark a point.</td>
<td>120</td>
</tr>
<tr>
<td>IVEdrawrectangle</td>
<td>Add a filled rectangle to an existing plot.</td>
<td>124</td>
</tr>
<tr>
<td>IVEerase</td>
<td>Remove all plots and reset the user graph.</td>
<td>121</td>
</tr>
<tr>
<td>IVEpause</td>
<td>Suspend the execution of a model.</td>
<td>122</td>
</tr>
<tr>
<td>IVEzoom</td>
<td>Scale the user graph.</td>
<td>123</td>
</tr>
</tbody>
</table>
**IVE_RGB**

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

**Synopsis**

```mosel
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:

```mosel
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**
Insert a new plot on the user graph.

**Synopsis**
```
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:
```
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.

**Synopsis**
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:

```mosel
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)
```

**Further information**
The arrow connects the two points whose coordinates are given as parameters, pointing to the second one.
**IVEdrawgantt**

**Purpose**
Draw a Gantt chart with CP operations.

**Synopsis**

```plaintext
procedure IVEdrawgantt(ops: array of cpoperation)
procedure IVEdrawgantt(ops: set of cpoperation)
```

**Argument**

| ops | An array or a set of CP operations. |

**Further information**

This procedure can only be used if the xpresscp module is present that defines the type `cpoperation` to represent operations (tasks) in scheduling and planning problems.

**Related topics**

See the Xpress-CP Reference Manual for further detail.
**IVEdrawlabel**

**Purpose**
Add a text box to an existing plot.

**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:

```mosel
...  
! 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)
```

**Further information**
The box will be centered horizontally just above the point given.
**IVEdrawline**

**Purpose**
Add a line to an existing plot.

**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.

```mosel
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)
```

**Further information**
The line connects the two points whose coordinates are given as parameters.
**IVEdrawpoint**

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

**Synopsis**
> 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:

```mosel
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**
Suspend the execution of a Mosel model at the line where the call occurs.

**Synopsis**
procedure IVEpause(message: string)

**Argument**
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**
Scale the user graph.

**Synopsis**

```plaintext
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.
**IVEdrawrectangle**

**Purpose**
Add a filled rectangle to an existing plot.

**Synopsis**

```plaintext
procedure IVEdrawrectangle(handle: integer, x1: real, y1: real, x2: real, y2: real)
```

**Arguments**

- `handle` The number returned by IVEaddplot.
- `x1` The x coordinate of the rectangle corner.
- `y1` The y coordinate of the rectangle corner.
- `x2` The x coordinate of the opposite rectangle corner.
- `y2` The y coordinate of the opposite rectangle corner.

**Example**

The following code draws a rectangle covering an area 10 units long and 1 unit high starting at the origin.

```
IVEdrawrectangle(plot,0,0,10,1)
```
**IVEdrawellipse**

**Purpose**
Add a filled ellipse to an existing plot.

**Synopsis**

```pascal
procedure IVEdrawellipse(handle: integer, x1: real, y1: real, x2: real, y2: real)
```

**Arguments**

- `handle` The number returned by IVEaddplot.
- `x1` The x coordinate of one corner of the ellipse's bounding rectangle.
- `y1` The y coordinate of one corner of the ellipse's bounding rectangle.
- `x2` The x coordinate of the opposite corner of the ellipse's bounding rectangle.
- `y2` The y coordinate of the opposite corner of the ellipse's bounding rectangle.

**Example**

The following code draws a very "flat" ellipse centered at the origin.

```pascal
IVEdrawellipse(plot,-5,0.5,5,-0.5)
```
Chapter 6
mmjava

The mmjava module for Mosel is intended for users who integrate their Mosel models into Java applications. To use this module, the following line must be included in the header of a Mosel model file:

```
uses 'mmjava'
```

Note that Mosel may fail loading mmjava if it is used from outside of a Java enabled application.

6.1 I/O drivers

This module provides the java and jraw IO drivers. The first one can be used to link a Mosel output (input) stream to a Java OutputStream (InputStream) or a Java ByteBuffer. The second driver is a modified version of the raw driver suitable for Java: instead of an address, this driver takes as input a reference to an object.

For both drivers, file names are replaced by references to objects. These references are of two kinds: direct references to public static objects (e.g. "java.lang.System.out") and names defined using the XPRM.bind method. The second technique will be used with non static objects: the method XPRM.bind establishes a link between a name and an object. This name can then be used as an object reference for mmjava drivers.

When using Java object from Mosel, it is important to make sure objects and related fields can be accessed: in particular the class and its fields must be public.

6.1.1 Driver java

```
java:static object|named object
```

With this driver a Java stream (OutputStream or InputStream) as well as a ByteBuffer can be used in place of a file in Mosel. This facility is specially useful for redirecting default Mosel streams to Java objects. Note that the Mosel Java interface uses this driver for redirecting default streams (in, out, and error) to the corresponding Java streams (System.in, System.out and System.err).

Example:

```
mosel=new XPRM();
mosel.bind("out", myout);  /* Associate 'myout' object with string "out" */
/* Redirect default output to 'myout' */
mosel.setDefaultStream(XPRM.F_OUTPUT|XPRM.F_LINBUF, "java:out")
/* Redirect error stream to Java output stream */
mosel.setDefaultStream(XPRM.F_ERROR, "java:java.lang.System.out")
```
### 6.1.2 Driver `jraw`

`jraw:[noindex,all]`

The driver can only be used in ‘initializations’ blocks. In the opening part of the block, no file name has to be provided, but general options can be stated at this point: they will be applied to all labels. Two options are supported:

- **all**: forces output of all cells of an array even if it is dynamic (by default only existing cells are considered).
- **noindex**: indicates that only data (no indices) are transferred between the Java objects and Mosel. By default, the first fields of each object are interpreted as index values for the array to be transferred. This behavior is changed by this option.

In the block, each label entry is understood as an object reference to use for the actual processing. Note that, before the object reference, one can add further options separated by commas, that are effective to the particular entry.

If the Model object to be initialized (or saved) is a scalar or an array with option `noindex`, the driver expects a Java object of a corresponding type (i.e. same basic type and scalar or one dimension array). If the option `noindex` is not used and the Mosel object is an array, the label must specify which fields of the class have to be taken into account for the mapping. This is indicated by a list of field names separated by commas and noted in brackets (e.g. "myobj(f1,f2,f3)").

In the following example the `jraw` driver is used to initialize an array of reals, `a`, and an array of integers, `ia`, with data held in the Java application that executes the model.

**Java part:**

```java
public class MyData {    // A class to store an 'array(string, int) of real'
    public String s; public int r; public double v;
    MyData(String i1, int i2, double v0) { s=i1; r=i2; v=v0; }
}
...  
MyData[] data;  
int[] intarr;  
...  
mosel=new XPRM();  
mosel.bind("data", data);    // Associate 'data' object with string "data"  
mosel.bind("ia", intarr);    // Associate 'intarr' object with string "ia"
```

**Mosel part:**

```mosel
declarations
a:array(string, range) of real
ia:array(range) of integer
end-declarations
...
initializations from "jraw:"
  aa as "data(s,r,v)" ! Initialize 'aa' with fields s,r,v of object 'data'
  ia as "noindex,ia" ! Initialize 'ia' with array 'ia'; no index (only values)
end-initializations
```
Thanks to this module it is possible to load several models in memory and execute them concurrently. A general synchronization mechanism based on event queues as well as two specialized IO drivers are also provided in order to ease the implementation of parallel algorithms in Mosel.

To use this module, the following line must be included in the header of the Mosel model file:

\[
\text{uses 'mmjobs'}
\]

### 7.1 Example

The following example shows how to compile, load, and then run a model from another model. After having started the execution, it waits for 60 seconds before stopping the secondary model if the latter has not yet finished.

```mosel
model "mmjobs example"
uses "mmjobs","mmsystem"

declarations
mymod: Model
event: Event
end-declarations

! Compile 'mymod.mos' to memory
if compile("","mymod.mos","shm:bin")<>0 then
  exit(1)
end-if

load(mymod,"shm:bin") ! Load bim file from memory...
fdelete("shm:bin") ! ... and release the memory block

! Disable model output
setdefstream(mymod,"","null:","null:")
run(mymod) ! Start execution and
wait(60) ! wait 1 min for an event

if isqueueempty then ! No event has been sent...
  writeln("Model too long: stopping it!")
  stop(mymod) ! ... stop the model then wait
  wait
end-if

! An event is available: model finished
event:=getnextevent
writeln("Exit status: ", getvalue(event))
writeln("Exit code : ", getexitcode(mymod))

unload(mymod)
end-model
```

---

**Chapter 7**

**mmjobs**
7.2 Procedures and functions

7.2.1 Model management

The type Model is used to reference a Mosel model. This section describes the procedures and functions available for model management: compilation of source model files, loading of bim files, execution and retrieval of model information. Note that before being used, a model has to be initialized by loading a bim file (load).

- **compile**  
  Compile a source model.  
  p. 130

- **getexitcode**  
  Get the exit code of a model.  
  p. 136

- **getid**  
  Get the ID of a model.  
  p. 134

- **getstatus**  
  Get the status of a model.  
  p. 135

- **load**  
  Load a Binary Model file.  
  p. 131

- **reset**  
  Reset a model.  
  p. 138

- **run**  
  Run a model.  
  p. 133

- **setdefstream**  
  Set default input/output streams of a model.  
  p. 132

- **stop**  
  Stop a running model.  
  p. 137

- **unload**  
  Unload a model.  
  p. 139
compile

**Purpose**

Compile a source model.

**Synopsis**

```plaintext
function compile(src:string):integer
function compile(opt:string, src:string):integer
function compile(opt:string, src:string, dst: string):integer
```

**Arguments**

- **opt**  
  Compilation options:
  - "g"  Include debugging information
  - "G"  Include tracing information
  - "s"  Strip symbols
  - "p"  Parse only: stop after the syntax analysis of the source file, do not compile (no file generated)
- **src**  Source file name
- **dst**  Destination file name

**Return value**

- 0  Function executed successfully
- 1  Parsing phase has failed (syntax error or file access error)
- 2  Error in compilation phase (a semantic error has been detected)
- 3  Error writing the output file
- 4  License error (compiler not authorized)

**Further information**

1. This function compiles a given model source file into a binary model file (.bim file) that is required as input to function `load` for executing the model.

2. If no destination file name is provided, the output file takes the same name as the source file with the extension `.bim`.

**Related topics**

- `load`
load

**Purpose**  
Load a Binary Model file.

**Synopsis**  
procedure load(mo:Model, bimf:string)

**Arguments**  
- mo Model object to be initialized  
- bimf Bim file name

**Further information**  
This procedure initializes the model mo with the bim file bimf. If mo has already been initialized, the model it references is unloaded before trying to load the new file (note that this operation fails if the model is running). If the file bimf cannot be accessed or one of the required modules cannot be loaded, the procedure generates an IO error (which may be intercepted if the control parameter ioctrl is true).

**Related topics**  
compile, setdefstream, run, unload.
**setdefstream**

**Purpose**
Set default input/output streams of a model.

**Synopsis**

```pascal
procedure setdefstream(mo:Model, wmd:integer, fname:string)
procedure setdefstream(mo:Model, input:string, output:string, error:string)
```

**Arguments**

- `mo` A Model
- `wmd` Stream to set. Possible values:
  - F_INPUT Default input stream
  - F_OUTPUT Default output stream
  - F_ERROR Default error stream
  - F_LINBUF Use line buffering
- `fname` Extended file name to be used for the stream.
- `input` Extended file name to be used for the input stream.
- `output` Extended file name to be used for the output stream.
- `error` Extended file name to be used for the error stream.

**Further information**

1. This function sets default IO streams to be used by a model. Model streams can be changed only when the model is not running. Each stream is associated to an extended file name (*i.e.* IO drivers can be used). For output streams, F_LINBUF may be specified (*e.g.* F_WRITE+F_LINBUF) in order to enable line buffering for the corresponding stream (the error stream is always open using line buffering).

2. For input and output streams, the filename is stored and streams are actually open when execution of the model starts: in case of an invalid file name, the error is not reported by this function. The error stream is immediately opened so in the case of an invalid file name is detected by this function.

3. Using an empty string as the file name implies resetting to the original default stream.
run

Purpose
Run a model.

Synopsis

procedure run(mo:Model)
procedure run(mo:Model, plist:string)

Arguments

mo Model to be executed
plist String composed of model parameter initializations separated by commas

Further information

1. This procedure starts the execution of a model in a new thread: when the procedure returns, the model is not necessarily started (this may be delayed depending on the operating system load) and not necessarily terminated (the second model is executing concurrently to the caller).

2. When the execution of the model is completed (normal termination, interruption after calling stop, or runtime error) or could not be started, an event of class EVENT_END is sent to the caller. The execution status is returned via the event value but it may also be obtained using getstatus. The exit code related to the last execution may be retrieved using getexitcode.

3. The specified model must have been previously initialized with load and must not be running. If the same model has to be executed several times concurrently, it must be loaded several times in different model objects.

Related topics
load, wait, stop, getstatus, getexitcode, reset.
getid

Purpose
Get the ID of a model.

Synopsis
function getid(mo:Model):integer

Argument
mo A model

Return value
ID of the model as an integer

Further information
Each model object has a unique ID number that can be obtained with this function. This ID may be used to identify the origin of an event (see getfromid).
getstatus

**Purpose**
Get the status of a model.

**Synopsis**
```plaintext
function getstatus(mo:Model):integer
```

**Argument**
- `mo` A model

**Return value**
- `RT_NOTINIT` Model has not been initialized or has been unloaded
- `RT_RUNNING` Model is running
- `RT_OK` Model is ready for execution and/or no error occurred during last execution
- `RT_MATHERR` A mathematical error occurred
- `RT_ERROR` A runtime error occurred
- `RT_IOERR` An IO error occurred
- `RT_STOP` Execution has been interrupted by a call to `stop`

**Related topics**
- `stop`, `getexitcode`
getexitcode

**Purpose**
Get the exit code of a model.

**Synopsis**
function getexitcode(mo:Model):integer

**Argument**
mo A model

**Return value**
Exit code of the last execution or 0

**Further information**
The exit code of the last execution corresponds to the value stated via a call to the procedure `exit`. The default exit value (i.e. procedure `exit` has not been called) is 0.

**Related topics**
`getstatus`.
stop

**Purpose**
Stop a running model.

**Synopsis**
```
procedure stop(mo:Model)
```

**Argument**
`mo` Model to interrupt

**Further information**
If the model is not currently running, no operation is performed. Note that the effect of this call may not be immediate and the corresponding model may continue running a few seconds before its effective interruption (for instance the time required to complete an IO operation).

**Related topics**
`run`. 
reset

Purpose
Reset a model.

Synopsis
procedure reset (mo:Model)

Argument
mo  Model to reset

Further information
This procedure resets a model after its execution: all resources it has allocated are released. The model returns to its state just after it has been loaded into memory. Note that this function is automatically called before a model is unloaded or run.

Related topics
run, unload.
unload

Purpose
Unload a model.

Synopsis
procedure unload(mo: Model);

Argument
mo Model to unload

Further information
This procedure unloads the given model. All resources used by this model, including modules, are released. The function fails if the model is running.

Related topics
load.
7.2.2 Synchronization

Synchronization between running models can be implemented using events. Events are characterized by a class and a value and may be exchanged between a model and its parent model. An event queue is attached to each model to collect all events sent to this model and is managed with a FIFO policy (First In – First Out). Depending on the needs, a model may check whether its queue is empty or simply suspend its execution until it has been sent an event.

The type Event represents an event in the Mosel language. Objects of type Event may be compared with = or <> and assigned with :=. The function nullevent returns an event without class and value: this is the initial value of a newly created event and no model can send an event of this kind (i.e. the class is necessarily not null).

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>dropnextevent</td>
<td>Drop the next event in the event queue of the model.</td>
<td>145</td>
</tr>
<tr>
<td>getclass</td>
<td>Get the class of an event.</td>
<td>149</td>
</tr>
<tr>
<td>getfromid</td>
<td>Get the ID of the sender of an event.</td>
<td>148</td>
</tr>
<tr>
<td>getnextevent</td>
<td>Get the next event in the event queue of the model.</td>
<td>144</td>
</tr>
<tr>
<td>getvalue</td>
<td>Get the value associated with an event.</td>
<td>150</td>
</tr>
<tr>
<td>isqueueempty</td>
<td>Check whether there are events waiting in the event queue.</td>
<td>146</td>
</tr>
<tr>
<td>nullevent</td>
<td>Return a ‘null’ event.</td>
<td>147</td>
</tr>
<tr>
<td>send</td>
<td>Send an event to a running model.</td>
<td>141</td>
</tr>
<tr>
<td>wait</td>
<td>Wait for an event.</td>
<td>142</td>
</tr>
<tr>
<td>waitfor</td>
<td>Wait for events of particular classes.</td>
<td>143</td>
</tr>
</tbody>
</table>
send

Purpose
Send an event to a running model.

Synopsis
procedure send(mo:Model, class:integer, value:real)
procedure send(class:integer, value:real)

Arguments
mo Model to send the event to
class Event class (must be >1)
value Event value

Further information
1. Events can be sent to models started by the caller (the child models) by using the first form of the procedure and to the model having started the caller (the parent model) with the second form of the procedure. An event can be received only by a running model using the mmjobs module: sending an event to a model that is not running or not using mmjobs is a no-operation.

2. Events are characterized by a class and a value. Event class values can be used to indicate the cause of the event (for instance, 2 could mean ‘a new solution has been found’) and the associated value may specify a property of the given instance (for example an objective value). Except for the special value 1 (EVENT_END) class values have no predefined meaning.

3. An event of class EVENT_END (=1) and model status as the event value is automatically sent by each model to its parent model when it terminates its execution.

Related topics
wait, waitfor.
wait

Purpose
Wait for an event.

Synopsis
procedure wait
procedure wait(dur:integer)

Argument
dur  A duration in seconds

Further information
This procedure suspends the execution of the caller until an event is available. The second form specifies a time limit: the processing is suspended for at most \texttt{dur} seconds.

Related topics
\texttt{send, waitfor, isqueueempty, getnextevent, dropnextevent}. 
waitfor

**Purpose**
Wait for events of particular classes.

**Synopsis**

```
procedure waitfor(mask:integer)
procedure waitfor(mask:integer,dur:integer)
```

**Arguments**

- `dur` A duration in seconds
- `mask` Bit mask of expected events

**Further information**

1. This procedure suspends the execution of the caller until an event of a particular class is available. The second form specifies a time limit: the processing is suspended for at most `dur` seconds.

2. The parameter `mask` is interpreted as a bit mask to select the expected events: all events sent to the model are automatically dropped until an event `ev` satisfies the following condition:

   \[
   \text{bittest(getclass(ev),mask)} <> 0
   \]

**Related topics**

`send`, `wait`, `isqueueempty`, `getnextevent`, `dropnextevent`.
getnextevent

**Purpose**
Get the next event in the event queue of the model.

**Synopsis**
```
function getnextevent:Event
```

**Return value**
The next event or `nullevent` if the queue is empty

**Further information**
The returned event is removed from the queue after it has been retrieved with this function.

**Related topics**
`dropnextevent`, `isqueueempty`.
**dropnextevent**

**Purpose**
Drop the next event in the event queue of the model.

**Synopsis**

```plaintext
procedure dropnextevent
```

**Further information**
This procedure has no effect if the event queue is empty.

**Related topics**
`getnextevent, isqueueempty`. 
isqueueempty

**Purpose**
Check whether there are events waiting in the event queue.

**Synopsis**
```
function isqueueempty:boolean
```

**Return value**
true if at least one event is available in the queue, false otherwise.

**Related topics**
dropnextevent, getnextevent.
nullEvent

**Purpose**
Return a ‘null’ event.

**Synopsis**
function nullevent:Event

**Return value**
An event of class and value equal to 0

**Further information**
Variables of type Event are initialized with this function.

**Related topics**
getnextevent.
**getfromid**

**Purpose**
Get the ID of the sender of an event.

**Synopsis**
```plaintext
function getfromid(ev:Event):integer
```

**Argument**
ev An event

**Return value**
The ID of the sender of the event (>0) or 0 for a `null` event

**Further information**
Each model has a unique ID that is attached to each event it sends. With this function one can identify the sender of a given event.

**Related topics**
`getid`, `getvalue`, `getclass`. 
**getclass**

**Purpose**
Get the class of an event.

**Synopsis**

```plaintext
function getclass(ev:Event):integer
```

**Argument**

- `ev` An event

**Return value**

The class of the event (>0) or 0 for a `null` event

**Further information**

A model sends automatically an event of class `EVENT_END`=1) when it terminates its processing. Other values are application specific.

**Related topics**

- `getvalue`
- `getfromid`
getvalue

**Purpose**
Get the value associated with an event.

**Synopsis**
```plaintext
function getvalue(ev:Event):real
```

**Argument**
- `ev` An event

**Return value**
The value of the event

**Further information**
In the case of an event of class `EVENT_END` (=1), this value corresponds to the model status.

**Related topics**
- `getclass`
- `getfromid`
7.3 I/O drivers

The *mmjobs* module provides a modified version of the *mem* IO driver designed to be used in a multithreaded environment: memory blocks allocated by the *shmem* IO driver are persistent (i.e. they are not released after the model terminates) and can be used by several models. Thanks to this facility, models running concurrently may exchange data through memory by means of initialization blocks for instance.

The driver *mempipe* offers another communication mechanism between models: a *memory pipe* may be open by two models simultaneously. One of them for writing and the other one for reading. This driver also supports *initialization blocks* through which data is transferred in binary form.

7.3.1 Driver *shmem*

*shmem:label[/minsize]*

The file name for this driver is a *label*: this is the identifier of the memory block. A label is not local to a particular model and remains valid after the end of the execution of the model having created it. All memory blocks are released when the module *mmjobs* is unloaded but a given memory block may also be deleted explicitly by calling the *fdelete* procedure of module *mmsystem* or by using the *fremove* C-function of the Native Interface.

Several models may open a given label at the same time and several read operations may be performed concurrently. However, writing to a memory block can be done by only one model at a time: if several models try to read and write from/to the same label, only one (it becomes the *owner* of the memory block) performs its IO operations for writing and the others are suspended until the owner closes its file descriptor to the specified label. Then, one of the waiting models is restarted and becomes the new owner: this process continues until all file descriptors to the label are closed.

Memory blocks are allocated dynamically and their size is increased automatically if necessary by pages of fixed size. When the file is closed, the system releases the memory that has not been used. In order to reduce memory fragmentation and increase efficiency (if the memory requirement is significantly larger than 4Kb), it is possible to specify an initial block size *minsize* (in bytes) when opening the file, *e.g.* for a block of 100Kb: "*shmem:mymem/102400*"

7.3.2 Driver *mempipe*

*mempipe:[all,][noindex,]name*

A *memory pipe* is characterized by its *name*. Only one model may open a pipe for reading but several models may open the same pipe for writing. However, if several models try to write to the same pipe, only one (it becomes the *owner* of the memory pipe) performs its IO operations and the others are suspended until the owner closes its file descriptor to the specified pipe. Then, one of the waiting models is restarted and becomes the new owner: this process continues until all file descriptors to the pipe are closed.

Pipe operations are possible only if the two ends of the pipe are open: one model for reading and at least one model for writing. There is no notion of ‘end of file’ in a pipe: if a model tries to read from an empty pipe (i.e. no model is writing to the other end) no error is raised and the model is suspended until something is available. Similarly trying to write to a pipe for which no model is reading from the other end is a blocking operation. In order to avoid lock ups, it is usually good practice to synchronize the models using events. For instance a model waits for a specific event before trying to read from a pipe: before starting to write to the same pipe, the other model sends the expected event.

Memory pipes may be used with initialization blocks. In this case, the pipe name can be completed by options *all* (all entries of dynamic arrays are transfered) and/or *noindex* (only values of array entries are transfered - indices do not precede).
This driver does not use labels for each record of the initialization block: it is assumed (but not checked) that both ends of the pipe are using the same sequence of records. For instance, if the writer sends an integer, a string and then an array of reals, the reader must expect an integer, a string and an array of reals: it is not allowed to skip records or change order as it is usually possible with these blocks.
Chapter 8

mmoci

The Mosel OCI (Oracle Call Interface) interface provides a set of procedures and functions that may be used to access Oracle databases. To use the OCI interface, the following line must be included in the header of a Mosel model file:

uses 'mmoci'

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

8.1 Example

Assume that the Oracle database contains a table “pricelist” of the following form:

<table>
<thead>
<tr>
<th>articlenum</th>
<th>color</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>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following small example shows how to logon to a database from an Mosel model file, read in data, and logoff from the database.

model 'OCIexample'
uses 'mmoci'
declarations
prices: array (range) of real
end-declarations
setparam("OCIverbose", true) ! Enable OCI message printing in case of error
OCIlogon("scott","tiger","") ! connect to Oracle as the user 'scott/tiger'
writeln("Connection number: ", getparam("OCIconnection"))

OCIExecute("select articlenum,price from pricelist", prices) ! Get the entries of field 'price' (indexed by ! field 'articlenum') in table 'pricelist'

OCIlogoff ! Disconnect from the database
end-model
Here the OCIverbose control parameter is set to true to enable OCI message printing in case of error. Following the connection, the procedure OCIexecute is called to retrieve entries from the field price (indexed by field articlenum) in the table pricelist. Finally, the connection is closed.

8.2 Data transfer between Mosel and Oracle

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

8.2.1 From Oracle to Mosel

Information is moved from Oracle 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
OCIexecute("SELECT c1,c2,c3 from T", mt)
```

behaves differently depending on the value of OCIndxcol. 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 mmoci:

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

With a table T containing:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>c2</td>
<td>c3</td>
</tr>
<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 OCIndxcol is false, all columns are treated as data. In this case, for each row (i,j,k) the following assignments are performed:

```mosel
mt(r,1):=i; mt(r,2):=j; mt(r,3):=k
```

where r is the row number in the result set.

Here, the resulting initialization is:

```mosel
mt(1,1)=1, mt(1,2)=2, mt(1,3)=5
mt(2,1)=4, mt(2,2)=3, mt(2,3)=6
```

The second argument of OCIexecute may also be a list of arrays. When using this version, the value of OCIndxcol 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:

```mosel
m1, m2: array(1..10) of integer
```

With the statement:

```mosel
OCIexecute("SELECT c1,c2,c3 from T", [m1,m2])
```
for each row \((i, j, k)\) of the result set, the following assignments are performed:

\[
m_1(i) := j; \quad m_2(i) := k
\]

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

\[
\begin{align*}
m_1(1) &= 2, \quad m_1(4) = 5 \\
m_2(1) &= 3, \quad m_2(4) = 6
\end{align*}
\]

### 8.2.2 From Mosel to Oracle

Information is transferred from Mosel to Oracle 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 their name in the expression. For instance in the following expression 3 parameters \((1, 2, 3)\) are used:

\[
\text{INSERT INTO } T \{c1, c2, c3\} \text{ VALUES } (:1, :2, :3)
\]

`mmoci` expects that parameters are always named \(:n\) where \(n\) is the parameter number starting at 1 but does not impose any order (i.e. \(:3, :1, :2\) is also valid) and a given parameter may be used several times in an expression. 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:

\[
mt: \text{array}(1..2, 1..3) \text{ of integer}
\]

and initialized with this assignment:

\[
mt::[1,2,3, 4,5,6]
\]

the execution of the call:

\[
\text{OCIexecute("INSERT INTO } T \{c1, c2, c3\} \text{ VALUES } (:1, :2, :3")}, mt)
\]

behaves differently depending on the value of `OCIndxcol`. If this control parameter is `true`, for each execution of the command, the following assignments are performed by `mmoci`:

\[
':1':= i, \quad ':2':= j, \quad ':3':= mt(i, j)
\]

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

\[
\begin{array}{ccc}
c_1 & c_2 & c_3 \\
1 & 1 & 1 \\
1 & 2 & 2 \\
1 & 3 & 3 \\
2 & 1 & 4 \\
2 & 2 & 5 \\
2 & 3 & 6
\end{array}
\]

Note that `mmoci` uses the names of the parameters to perform an initialization and not their relative position. This property is particularly useful for `UPDATE` statements where the order of parameters needs to be changed. For instance, if we want to update the table \(T\) instead of inserting new rows, we can write:

\[
\text{OCIexecute("UPDATE } T \text{ set } c3 = :3 \text{ WHERE } c1 = :1, \, c2 = :2"}, mt)
\]

This command is executed exactly in the same way as the `INSERT` example above (i.e. we do not have \(':3':=i, \quad ':1':=j, \quad ':2':=mt(i, j)\) as the order of appearance in the command suggests but \(':1':=i, \quad ':2':=j, \quad ':3':=mt(i, j)\)).
The same functionality may also be used to reorder or repeat columns. With the same definition of the array mt as before and a 4-column table S in the database the execution of the command

```
OCIexecute("INSERT INTO S (c1,c2,c3,c4) VALUES (:1,:2,:3,:2)",mt)
```

results in the following contents of table S:

<table>
<thead>
<tr>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>

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

```
':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:

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

When **OCIexecute** is used with a list of arrays, the behavior is again similar to what has been described earlier for the **SELECT** command: the first parameter(s) are assigned index values and the final ones the actual array values. For instance, assuming we have the following declarations:

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

And the arrays have been initialized as follows:

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

Then the following call:

```
OCIexecute("INSERT INTO T (c1,c2,c3) VALUES (:1,:2,:3)",[ml,m2])
```

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

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

The resulting table T is therefore:

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

### 8.3 Control parameters

The following parameters are defined by **mmoci**:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIautocommit</td>
<td>Enable/disable &quot;commit on success&quot; in OCI.</td>
<td>157</td>
</tr>
<tr>
<td>OCIBufsize</td>
<td>Data buffer size.</td>
<td>157</td>
</tr>
</tbody>
</table>
OCIcolsize   Maximum string length.  p. 157
OCIconnection Identification number of the active OCI connection.  p. 158
OCIdebug Enable/disable debug mode.  p. 158
OCIIndxcol Indicate whether to use first columns as indices.  p. 158
OCIrrowcnt Number of lines affected by the last SQL command.  p. 159
OCIrrowxfr Number of lines transferred during the last SQL command.  p. 159
OCIsuccess Indicate whether the last SQL command succeeded.  p. 159
OCIverbose Enable/disable message printing by OCI.  p. 159

All parameters can be accessed with the Mosel function getparam, and those that are not marked read-only in the list below may be set using the procedure setparam.

Example:

setparam("OCIverbose", true)  ! Enable message printing by OCI
csize:=getparam("OCIcolsize")  ! Get the maximum string length
setparam("OCIconnection", 3)  ! Select the connection number 3

---

**OCIautocommit**

**Description**  Enable/disable “commit on success” in OCI.

**Type**  Boolean, read/write

**Values**

- **true**  Changes to the database are committed automatically.
- **false**  Transactions have to be explicitly committed (or rolled back) using OCIcommit (or OCIrollback).

**Default value**  true

---

**OCIbufsize**

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

**Type**  Integer, read/write

**Values**  At least 1

**Default value**  4

**Affects routines**  OCIexecute, OCIreadstring.

---

**OCIcolsize**

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

**Type**  Integer, read/write
### OCIconnection

**Description**
Identification number of the active OCI connection. By changing the value of this parameter, it is possible to work with several connections simultaneously.

**Type**
Integer, read/write

**Affects routines**
OCIlogoff, OCIexecute, OCIconnection, OCIreadinteger, OCIreadreal, OCIreadstring.

**Set by routines**
OCIlogon.

### OCIdebug

**Description**
When this parameter is set to true, OCIverbose is also enabled and any SQL request sent to Oracle is displayed to the error stream before execution.

**Type**
Boolean, read/write

**Values**
- true  Enable debug mode.
- false Disable debug mode.

**Default value**
false

**See also**
OCIverbose.

### OCIdxcol

**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, for example, if one is trying to access a non-relational table, perhaps a dense table. Note this mode can be enabled only if at least the last dimension of each array is of fixed size.

**Type**
Boolean, read/write

**Values**
- true  Interpret the first columns of each row as indices.
- false Do not interpret the first columns of each row as indices.

**Default value**
true

**Affects routines**
OCIexecute, OCIreadinteger, OCIreadreal, OCIreadstring.
OCIrowcnt

Description  Number of lines affected by the last SQL command.
Type  Integer, read only
Set by routines  OCIexecute, OCIreadinteger, OCIreadreal, OCIreadstring.
See also  OCIrowxfr.

OCIrowxfr

Description  Number of lines transferred during the last SQL command.
Type  Integer, read only
Set by routines  OCIexecute, OCIreadinteger, OCIreadreal, OCIreadstring.
See also  OCIrowcnt.

OCIsuccess

Description  Indicate whether the last SQL command has been executed successfully.
Type  Boolean, read only
Values  
true  Success.
false  Error.
Set by routines  All OCI functions.

OCIverbose

Description  Enable/disable message printing by OCI.
Type  Boolean, read/write
Values  
true  Enable message printing.
false  Disable message printing.
Default value  true

8.4 Procedures and functions

This section lists in alphabetical order the functions and procedures that are provided by the mmoci module.

OCIcommit  Commit the current transaction.  p. 167
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCIexecute</td>
<td>Execute an SQL command.</td>
<td>163</td>
</tr>
<tr>
<td>OCIlogoff</td>
<td>Terminate the active database connection.</td>
<td>162</td>
</tr>
<tr>
<td>OCIlogon</td>
<td>Connect to a database.</td>
<td>161</td>
</tr>
<tr>
<td>OCIreadinteger</td>
<td>Read an integer value from a database.</td>
<td>164</td>
</tr>
<tr>
<td>OCIreadreal</td>
<td>Read a real value from a database.</td>
<td>165</td>
</tr>
<tr>
<td>OCIreadstring</td>
<td>Read a string from a database.</td>
<td>166</td>
</tr>
<tr>
<td>OCIrollback</td>
<td>Roll back the current transaction.</td>
<td>168</td>
</tr>
</tbody>
</table>
OCIlogon

Purpose
Connect to a database.

Synopsis
procedure OCIlogon(s:string)
procedure OCIlogon(u:string,p:string,db:string)

Arguments
  s  Logon string as "user/password@db"
  n  User name
  p  Password
  db Database name (may be "" for the default database)

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

   OCIlogon("yves/DaSH@test")

Open a connection to the default database the user ‘scott’ with the password ‘tiger’

   OCIlogon("scott","tiger","")

Further information
This procedure establishes a connection to the database db as user n/p. It is possible to open several connections but the connection established last becomes active. Each connection is assigned an identification number which can be obtained by getting the value of the parameter OCIconnection after this procedure has been executed. This parameter can also be used to change the active connection.

Related topics
OCIlogoff.
OCIlogoff

**Purpose**
Terminate the active database connection.

**Synopsis**
procedure OCIlogoff

**Further information**
The active connection can be accessed or changed by setting the control parameter OCIconnection.

**Related topics**
OCIlogon.
OCIexecute

Purpose
Execute an SQL command.

Synopsis
procedure OCIexecute(s:string)
procedure OCIexecute(s:string, a:array)
procedure OCIexecute(s:string, l:list)
procedure OCIexecute(s:string, m:set)

Arguments
s  SQL command to be executed
a  An array
l  A list. May be a list of arrays
m  A set

Example
The following example contains four OCIexecute statements performing the following tasks:

- Get all different values of the column color in the table pricelist.
- Initialize the arrays colors and prices with the values of the columns color and price of the table pricelist.
- Create a new table newtab in the active database with 2 columns, ndx and price.
- Add data entries to table newtab.

    declarations
    prices: array(1001..1004) of real
    colors: array(1001..1004) of string
    allcolors: set of string
    end-declarations

    OCIexecute("select color from pricelist", allcolors)
    OCIexecute("select articlenum,color,price from pricelist", [colors,prices])
    OCIexecute("create table newtab (ndx integer, price double)")
    OCIexecute("insert into newtab (ndx, price) values (:1,:2)", prices)

Further information
1. This procedure executes the given SQL command. The user is referred to the Oracle documentation for further information on PL/SQL.

2. For output commands (like insert into) this procedure accepts arrays, sets and lists of basic types (integer, real, string or Boolean) as well as module types for which from/to string conversions are available. Record types composed of scalars or other records can also be used (the fields that cannot be handled are silently ignored). It is also possible to use a list arrays of basic types (all arrays must be indexed by the same sets) or a list of scalar elements of different basic or module types.

3. For input commands (like select from) the same restrictions apply for arrays and list of arrays but lists and sets must be of a basic type.

Related topics
OCIreadinteger, OCIreadreal, OCIreadstring.
OCIreadinteger

**Purpose**
Read an integer value from a database.

**Synopsis**
function OCIreadinteger(s:string):integer

**Argument**
s SQL command for selecting the value to be read

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

**Example**
The following gets the article number of the first data item in table pricelist for which the field color is set to blue:

```
i:=OCIreadinteger("select articlenum from pricelist where color=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**
OCIexecute, OCIreadreal, OCIreadstring.
OCIreadreal

**Purpose**
Read a real value from a database.

**Synopsis**
function OCIreadreal(s:string):real

**Argument**
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 table newtab:

```
r:=OCIreadreal("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**
OCIexecute, OCIreadinteger, OCIreadstring.
**OCIreadstring**

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

**Synopsis**

```function OCIreadstring(s:string):string```

**Argument**

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

**Return value**

String read or empty string.

**Example**

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

```s:=OCIreadstring("select color from pricelist where articlenum=1004")```

**Further information**

1. The empty string is returned if no real 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**

- `OCIexecute`, `OCIreadinteger`, `OCIreadreal`.
OCIcommit

Purpose
Commit the current transaction.

Synopsis
procedure OCIcommit

Further information
This procedure is required only if the control parameter OCIautocommit is set to false.

Related topics
OCIrollback.
OCIrollback

**Purpose**  
Roll back the current transaction.

**Synopsis**  
procedure OCIrollback

**Further information**  
This procedure can be used only if the control parameter OCIautocommit is set to false.

**Related topics**  
OCIcommit.
8.5 I/O drivers

This module provides a driver designed to be used in initializations blocks for both reading and writing data. The oci IO driver simplifies access to Oracle databases.

8.5.1 Driver oci

oci: [debug; ] [noindex; ] logstring

The driver can only be used in 'initializations' blocks. The database to use has to be given in the opening part of the block as user/password@dbname. Before this identifier, the following options may be stated:

- debug to execute the block in debug mode (to display what SQL queries are produced),
- noindex to indicate that only data (no indices) are transferred between the data source and Mosel. By default, the first columns of each table is interpreted as index values for the array to be transferred. This behaviour is changed by this option.
- colsize=c to set the size of a text column (default 64 characters),
- bufsize=c to set the size of the data buffer in kilobytes (default 4).

In the block, each label entry is understood as a table name optionally followed by a list of column names in brackets (e.g. "my_table(col1,col2)"). All columns are used if no list of names is specified. Note that, before the table name, one can add option noindex to indicate that for this particular entry indices are not used.

Example:

```mosel
initializations from "mmoci.oci:scott/tiger@orcl"
NWeeks as "PARAMS(Weeks)"  ! Initialize 'NWeeks' with column 'Weeks'
BPROF as "noindex;BPROFILE"  ! Initialize 'BPROF' with table 'BPROFILE'
end-initializations
```

Mosel Reference Manual
Chapter 9
mmodbc

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.

9.1 Prerequisite

The ODBC technology relies on a driver manager that is used as an interface between applications (like mmodbc) and a data source itself accessed through a dedicated driver. As a consequence, this module requires that both, a driver manager and the necessary drivers (one for each data source to be used), are installed and set up on the operating system.

Under Windows, usually the driver manager is part of the system and most data sources are provided with their ODBC driver (for instance Excel, Access or SQLServer).

On the other supported operating systems it may be necessary to install a driver manager (as well as the necessary drivers). The module mmodbc supports two driver managers: iODBC (http://www.iodbc.org) and unixODBC (http://www.unixodbc.org). Upon startup the module tries to load the dynamic library "libiodbc.so" ("libiodbc.sl" under HP-UX) then, if this fails, tries "libodbc.so" ("libodbc.sl" under HP-UX). The initialization succeeds only if one of these libraries can be found and publishes the required symbols. In case of initialization failure, please make sure that one of the driver managers is installed and that the corresponding libraries can be accessed (in general this requires updating some environment variable).

9.2 Example

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>color</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>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The following small example shows how to connect to a database from an Mosel model file, read in data, and disconnect from the database.

```mosel
model 'ODBCexample'
uses 'mmodbc'
declarations
   prices: array (range) of real
end-declarations
setparam("SQLverbose", true) ! Enable ODBC message printing in case of error
SQLconnect("DSN=mydata") ! Connect to the database defined by 'mydata'
writeln("Connection number: ", getparam("SQLconnection"))
SQLexecute("select articlenum,price from pricelist", prices)
   ! Get the entries of field 'price' (indexed by 
   ! field 'articlenum') in table 'pricelist'
SQLdisconnect ! Disconnect from the database
end-model
```

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.

### 9.3 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.

#### 9.3.1 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:

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

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:

\[
\begin{align*}
mt(1,1)=1, & \quad mt(1,2)=2, \quad mt(1,3)=5 \\
mt(2,1)=4, & \quad mt(2,2)=3, \quad mt(2,3)=6
\end{align*}
\]

The second argument of \texttt{SQLexecute} may also be a list of arrays. When using this version, the value of \texttt{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:

\[
ml, \ m2: \text{array}(1..10) \text{ of integer}
\]

With the statement:

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

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

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

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

\[
\begin{align*}
ml(1)=2, & \quad m1(4)=5 \\
m2(1)=3, & \quad m2(4)=6
\end{align*}
\]

9.3.2 From Mosel to the database

Information is transferred from Mosel to the database when performing an \texttt{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:

\[
\text{INSERT INTO T (c1,c2,c3) VALUES (?,?,?)}
\]

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 \texttt{SELECT} statement.

Assuming \( mt \) has been declared as follows:

\[
mt: \text{array}(1..2,1..3) \text{ of integer}
\]

and initialized with this assignment:

\[
mt::[1,2,3, \quad 4,5,6]
\]

the execution of the call:

\[
\text{SQLexecute("INSERT INTO T (c1,c2,c3) VALUES (?,?,?),mt")}
\]

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

\[
'?1':= i, \quad '?2':= j, \quad '?3':= mt(i,j)
\]

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

```sql
'?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:

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

When SQLexecute is used with a list of arrays, the behavior is again similar to what has been described earlier for the SELECT command: the first parameter(s) are assigned index values and the final ones the actual array values. For instance, assuming we have the following declarations:

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

And the arrays have been initialized as follows:

```mosel
ml::[1,2,3]
m2::[4,5,6]
```

Then the following call:

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

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

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

The resulting table \( T \) is therefore:

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

## 9.4 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 to 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 type of columns is deduced by the Excel driver by scanning the first 8 rows. The
number of rows analyzed can be changed using the option \texttt{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 is added using an \texttt{INSERT} statement. To overwrite existing data in the worksheet, the SQL statement \texttt{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 has already been written to a range using an \texttt{INSERT} statement. Within Excel it is not sufficient to delete the previous data by selecting it and hitting the Delete key. If this is done, further data will be added after a blank rectangle where the deleted data used to reside. Instead, it is important to use Edit/Delete/Shift cells up within Excel, which will eliminate all traces of the previous data, and the enlarged range.

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

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.

## 9.5 Control parameters

The following parameters are defined by \texttt{mmodbc}:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLbufsize</td>
<td>Data buffer size.</td>
<td>175</td>
</tr>
<tr>
<td>SQLcolsize</td>
<td>Maximum string length.</td>
<td>175</td>
</tr>
<tr>
<td>SQLconnection</td>
<td>Identification number of the active ODBC connection.</td>
<td>175</td>
</tr>
<tr>
<td>SQLdebug</td>
<td>Enable/disable debug mode.</td>
<td>175</td>
</tr>
<tr>
<td>SQLdm</td>
<td>Driver manager currently used.</td>
<td>176</td>
</tr>
<tr>
<td>SQLextn</td>
<td>Enable/Disable extended syntax.</td>
<td>176</td>
</tr>
<tr>
<td>SQLndxcol</td>
<td>Indicate whether to use first columns as indices.</td>
<td>176</td>
</tr>
<tr>
<td>SQLrowcnt</td>
<td>Number of lines affected by the last SQL command.</td>
<td>176</td>
</tr>
<tr>
<td>SQLrowxfr</td>
<td>Number of lines transfered during the last SQL command.</td>
<td>177</td>
</tr>
<tr>
<td>SQLsuccess</td>
<td>Indicate whether the last SQL command succeeded.</td>
<td>177</td>
</tr>
<tr>
<td>SQLverbose</td>
<td>Enable/disable message printing by the ODBC driver.</td>
<td>177</td>
</tr>
</tbody>
</table>

All parameters can be accessed with the Mosel function \texttt{getparam}, and those that are not marked read-only in the list below may be set using the procedure \texttt{setparam}.

Example:

```mosel
setparam("SQLverbose", true) ! Enable message printing by the ODBC driver
csize:=getparam("SQLcolsize") ! Get the maximum string length
setparam("SQLconnection", 3)! Select the connection number 3
```
### 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 value**  
4

**Affects routines**  
SQLexecute, SQLreadstring.

### SQLcolsize

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

**Type**  
Integer, read/write

**Values**  
Between 8 and 1024

**Default value**  
64

**Affects routines**  
SQLexecute, SQLreadstring.

### SQLconnection

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

**Type**  
Integer, read/write

**Affects routines**  
SQLdisconnect, SQLexecute, SQLreadinteger, SQLreadreal, SQLreadstring.

**Set by routines**  
SQLconnect.

### SQLdebug

**Description**  
When this parameter is set to `true`, SQLverbose is also enabled and any SQL request sent to ODBC is displayed to the error stream before execution.

**Type**  
Boolean, read/write

**Values**  
`
true  
false
`

**Default value**  
false

**See also**  
SQLverbose.
### SQLdm

**Description**: Driver manager currently used.

**Type**: Integer, read only

**Values**:
- 0: Unspecified (manager not loaded dynamically).
- 1: iODBC.
- 2: unixODBC.

### SQLextn

**Description**: Enable/Disable extended syntax.

**Type**: Boolean, read/write

**Values**:
- true: Enable extended syntax.
- false: Disable extended syntax.

**Default value**: true

### 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, for example, if one is trying to access a non-relational table, perhaps a dense spreadsheet table. Note this mode can be enabled only if at least the last dimension of each array is of fixed size.

**Type**: Boolean, read/write

**Values**:
- true: Interpret the first columns of each row as indices.
- false: Do not interpret the first columns of each row as indices.

**Default value**: true

**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.

**See also**: SQLrowxfr.
### SQLrowxfr

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

**Type**
Integer, read only

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

**See also**
SQLrowcnt.

---

### SQLsuccess

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

**Type**
Boolean, read only

**Values**
- true  
  Success.
- false 
  Error.

**Set by routines**
All ODBC functions.

---

### SQLverbose

**Description**
Enable/disable message printing by the ODBC driver.

**Type**
Boolean, read/write

**Values**
- true  
  Enable message printing.
- false 
  Disable message printing.

**Default value**
true

---

# 9.6 Procedures and functions

This section lists in alphabetical order the functions and procedures that are provided by the mmodbc module.

- **SQLconnect**
  Connect to a database.  
  p. 178

- **SQLdisconnect**
  Terminate the active database connection.  
  p. 179

- **SQLexecute**
  Execute an SQL command.  
  p. 180

- **SQLreadinteger**
  Read an integer value from a database.  
  p. 181

- **SQLreadreal**
  Read a real value from a database.  
  p. 182

- **SQLreadstring**
  Read a string from a database.  
  p. 183

- **SQLupdate**
  Update the selected data with the provided array(s).  
  p. 184
SQLconnect

**Purpose**
Connect to a database.

**Synopsis**
procedure SQLconnect(s:string)

**Argument**
s   Connection string

**Example**
The following connects to the MySQL database 'test' as the user 'yves' with the password 'DaSH':

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

Open a connection to Excel using the file *mydata.xls*

```plaintext
SQLconnect("mydata.xls")
```

**Further information**
This procedure establishes a connection to the database defined by the given connection string. If extended mode is in use (default) and the ODBC driver manager publishes its driver list, the connection string may be reduced to a file name as long as this name allows identification of the required driver (by using the filename extension). It is possible to open several connections but the connection established last becomes active. Each connection is assigned an identification number which can be obtained by getting the value of the 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 accessed or 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, l:list)
procedure SQLexecute(s:string, m:set)

Arguments
s SQL command to be executed
a An array
l A list
m A set

Example
The following example contains four SQLexecute statements performing the following tasks:

• Get all different values of the column color in the table pricelist.
• Initialize the arrays colors and prices with the values of the columns color and price of the table pricelist.
• Create a new table newtab in the active database with 2 columns, ndx and price.
• Add data entries to table newtab.

declarations
prices: array(1001..1004) of real
colors: array(1001..1004) of string
allcolors: set of string
end-declarations

SQLexecute("select color from pricelist", allcolors)
SQLexecute("select articlenum,color,price from pricelist", [colors,prices])
SQLexecute("create table newtab (ndx integer, price double)")
SQLexecute("insert into newtab (ndx, price) values (?,?)", prices)

Further information
1. This procedure executes the given SQL command. The user is referred to the documentation of the database driver he is using for more information about the commands that are supported by it. Note that if extended syntax is in use (default), parameters usually noted '?' in normal SQL queries may be numbered (like '?1','?2',...) in order to control in which order are mapped columns of data source table to Mosel arrays. This feature is especially useful when writing 'update' queries for which indices must appear after values (e.g. "update mytable set datacol=?2 where ndxcol=?1").

2. For output commands (like insert into) this procedure accepts arrays, sets and lists of basic types (integer, real, string or Boolean) as well as module types for which from/to string conversions are available. Record types composed of scalars or other records can also be used (the fields that cannot be handled are silently ignored). It is also possible to use a list arrays of basic types (all arrays must be indexed by the same sets) or a list of scalar elements of different basic or module types.

3. For input commands (like select from) the same restrictions apply for arrays and list of arrays but lists and sets must be of a basic type.

Related topics
SQLUpdate, SQLreadinteger, SQLreadreal, SQLreadstring.
SQLreadinteger

**Purpose**
Read an integer value from a database.

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

**Argument**
s
SQL command for selecting the value to be read

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

**Example**
The following gets the article number of the first data item in table pricelist for which the field color is set to blue:

```
i:=SQLreadinteger("select articlenum from pricelist where color=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 a database.

**Synopsis**
function SQLreadreal(s:string):real

**Argument**
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 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 a database.

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

**Argument**
s      SQL command for selecting the string to be read

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

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

s:=SQLreadstring("select color from pricelist where articlenum=1004")

**Further information**
1. The empty string is returned if no real 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 list of arrays

**Example**
The following example initializes the array `prices` with the values of the table `pricelist`, changes some values in the array and finally, updates the date 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.
9.7 I/O drivers

This module provides two drivers designed to be used in initializations blocks for both reading and writing data. The odbc IO driver simplifies access to ODBC enabled data sources and the excel IO driver is a direct interface to the MS Excel spreadsheet. The excel driver is available on the Windows platform only.

9.7.1 Driver odbc

The driver can only be used in ‘initializations’ blocks. The Data Source Name to use has to be given in the opening part of the block. Before the DSN, the following options may be stated:

- **debug** to execute the block in debug mode (to display what SQL queries are produced),
- **noindex** to indicate that only data (no indices) are transferred between the data source and Mosel. By default, the first columns of each table is interpreted as index values for the array to be transferred. This behaviour is changed by this option,
- **colsize=c** to set the size of a text column (default 64 characters),
- **bufsize=c** to set the size of the data buffer in kilobytes (default 4).

In the block, each label entry is understood as a table name optionally followed by a list of column names in brackets (e.g. "my_table(col1, col2)"). All columns are used if no list of names is specified. Note that, before the table name, one can add option noindex to indicate that for this particular entry indices are not used.

Example:

```
initializations from "mmodbc.odbc:auction.xls"
NWeeks as "PARAMS(Weeks)" ! Initialize 'NWeeks' with column 'Weeks'
! of table 'PARAMS'
BPROF as "noindex;BPROFILE" ! Initialize 'BPROF' with table 'BPROFILE'
! all columns being data (no indices)
end-initializations
```

9.7.2 Driver excel

The driver can only be used in ‘initializations’ blocks. The file name to use (an Excel spreadsheet with the .xls extension) has to be given in the opening part of the block. Before this name, four options may be stated:

- **noindex** indicates that only data (no indices) are transferred between the spreadsheet and Mosel. By default, the first columns of each table is interpreted as index values for the array to be transferred. This behaviour is changed by this option
- **grow** when writing data, the driver uses the provided range ignoring the end of the data if there is not enough space. When this option is specified, the driver extends the range by adding lines if necessary
- **skiph** with this option, the driver skips the first line (or header) of the provided range. If the range contains only one line, the following line is selected
- **newx** the driver does not open the file if it can find a running instance of Excel having the required file open: it works directly with the application and modifications made to the workbook are not saved when the file is closed in Mosel. If this option is specified a new instance of Excel is started in all cases and the workbook is saved before quitting the application when the file is closed in Mosel.
In the block, each label entry is understood as a range in the workbook: named ranges are represented by their name (e.g. "MyRange") and explicit ranges are noted using square brackets (e.g. "[sheet1$al:c2]"). For explicit ranges, the sheet is identified by its name or number and separated from the cell selection with the $ sign. The first sheet of the workbook is selected if no indication is given. Similarly, the first cell of the selected sheet is assumed if no selection is provided ("[]" is the same as "[1$al]"). Note that, before the range selection, one can add options as for the file opening. For instance, "skip;grow;" is suitable to use a named range formatted for an ODBC connection.

Example:

```
initializations from "mmodbc.excel:skip; auction.xls"
NWeeks as "[b1:d12]"  ! Initialize 'NWeeks' with data in b2:d12
BPROF as "noindex; BPROFILE"  ! Initialize 'BPROF' with named range 'BPROFILE'
                              ! all columns being data (no indices)
end-initializations
```
Chapter 10

mmquad

The mmquad module 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:

uses 'mmquad'

The first section 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 section. By means of an example it is shown how the functions published by mmquad can be used in another module for accessing quadratic expressions and working with them.

10.1 New functionality for the Mosel language

10.1.1 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).

10.1.1.1 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). The following

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
\[ 3x(1) - 2x(3) - x(4) \leq 100 \]
\[ x(1) + 3x(2) + 3x(3) - 2x(4) \Rightarrow 10 \]
\[ x(1) + 3x(2) + 3x(3) - 2x(4) \leq 30 \]

\[ 2 \leq x(1); x(1) \leq 20 \]
\[ x(2) \text{ is integer}; x(3) \text{ is integer} \]
\[ x(4) \text{ is free} \]

! The objective function is a quadratic expression
\[ \text{Obj} := x(1) + x(1)^2 + 2x(1)x(2) + 2x(2)^2 + x(4)^2 \]

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

### 10.1.2 Procedures and functions

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

- `exportprob` Export a quadratic problem to a file. p. 189
- `getsol` Get the solution value of a quadratic expression. p. 190
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
EP_MAX   LP format, maximization
EP_MPS   MPS format
EP_STRIP Use scrambled names
filename Name of the output file; if empty, output printed to standard output (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 the file prob1.lp maximizing constraint Profit:

uses "mmquad"
declarations
  Profit:qexp
end-declarations
...
exportprob(0, ",", Profit)
exportprob(EP_MAX, "prob1", 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, Mosel appends .lp to it for LP
format files and .mat for MPS format.
getsol

Purpose
Get the solution value of a quadratic expression.

Synopsis
function getsol(q:qexp):real

Argument
q
A quadratic expression

Return value
Solution value or 0.

Example
uses "mmquad"
declarations
x,y,z: mpvar
Profit:qexp
end-declarations
... ! (Define and solve the problem)
writeln("Profit value: ", getsol(Profit))
writeln("Evaluation of an expression: ", getsol(x*y+5*z^2))

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.
10.2 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;
XPRMsosolib dso;
mmquad_imci mq;
void **quadctx;

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

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

10.2.1 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).

```c
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.

```c
#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}
};
static const char *deplist[] = {"mmquad", NULL}; /* Module dependency list */

/* 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(XPRMdsoserv), 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;

    dso = mm->finddso("mmquad"); /* Retrieve reference to the mmquad module */
    quadctx = *(mm->getdsoctx(ctx, dso, (void **)(&mq)));
    /* Get the module context and the communication interface of mmquad */
    q = XPRM_POP_REF(ctx); /* Get the quadratic expression from the stack */

    /* 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", 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");
}
10.2.2 Description of the library functions

- **clearqexpstat**: Free the memory allocated by getqexpstat. p. 196
- **getqexpnextterm**: Enumerate the terms of a quadratic expression. p. 197
- **getqexpso1**: Evaluate a quadratic expression. p. 194
- **getqexpstat**: Get information about a quadratic expression. p. 195
**getqexpsol**

**Purpose**  
Return an evaluation of a quadratic expression based on the current solution.

**Synopsis**  
```c
double getqexpsol(XPRMctx ctx, void *quadctx, mmquad_qexp q);
```

**Arguments**  
- `ctx`  Mosel's execution context  
- `quadctx`  Context of `mmquad`  
- `q`  Reference to a quadratic expression

**Return value**  
An evaluation of the expression on the current solution.

**Further information**  
This function returns an evaluation of a quadratic expression based on last solution obtained from the optimizer. This is the function called when using `getsol` on a quadratic expression from a Mosel program.
getqexpstat

Purpose
Get information about a quadratic expression.

Synopsis
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 change flag is returned (may be NULL). Possible values of this flag:
  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

Further information
A call to this function frees the table of variables that has previously been allocated by a call to function getqexpstat.

Related topics
getqexpstat.
**getqexpnextterm**

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

**Synopsis**
```c
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.

**Example**
The following displays the terms of a quadratic expression:
```c
void dispqexp(XPRMcontext ctx, mmquad_qexp q)
{
    void *prev;
    XPRMmpvar v1,v2;
    double coeff;
    int nlin,ct;

    mq->getqexpstat(ctx, quadctx, q, &nlin, NULL, NULL, NULL);
    ct=0;
    prev=mq->getqexpnextterm(ctx, quadctx, q, NULL, &v1, &v2, &coeff);
    mm->printf(ctx, "%g ", coeff);
    while(prev!=NULL) {
        prev=mq->getqexpnextterm(ctx, quadctx, q, prev, &v1, &v2, &coeff);
        if(ct<nlin) { mm->printf(ctx,"%g %p", coeff, v2); ct++; }
        else mm->printf(ctx,"%g %p * %p", coeff, v1, v2);
    }
    mm->printf(ctx,\"\n\");
}
```

**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. Note that the behavior of these operators may vary between systems. To use this module, the following line must be included in the header of the Mosel model file:

```
uses 'mmsystem'
```

11.1 New functionality for the Mosel language

11.1.1 The type text

This module provides the type `text` for text manipulation. Like the Mosel basic type `string`, this new type may be generated from all objects that can be converted to a text representation and supports the usual string operations (like concatenation or formatting). In addition, text objects can be altered (one can get and change a single as well as a sequence of characters in a text); offer a wider set of operations (like insertion/deletion/search of substrings) and, as all module types, are passed by reference to subroutines.

11.1.2 The type date

As the name suggests, the type `date` is used to represent a calendar date. Internally, a date is stored as three independent integers for representing the year (-32768 to 32767), the month (-128 to 127) and the day in the month (-128 to 127). The validity of a date can be checked using the function `isvalid`. A date object can be initialized by a text string, a single or three numerical values. In the first case, the conversion is processed using a predefined date format (see `datefmt`); in the second case, the integer is interpreted as the number of days elapsed since 1/1/1970; finally, if three integers are used, they are respectively interpreted as the year, month and day for the date. The constant `SYS_NOW` may also be used to initialize a date: `date(SYS_NOW)` is the current date. This type also supports assignment, comparison as well as difference (returned in number of days) and addition/subtraction of an integer (number of days).

11.1.3 The type time

The type `time` is used to represent a time during the day. Internally, a time object is stored as an integer representing a number of milliseconds. A time object can be initialized by a text string or one to four numerical values. In the first case, the conversion is processed using a predefined time format (see `timefmt`); in the second case, the integer is interpreted as a number of milliseconds. When two to four integers are used, they are understood as the hours, minutes, seconds and milliseconds. The constant `SYS_NOW` may also be used to initialize a time: `time(SYS_NOW)` is the current time. This type also supports assignment, comparison as well as difference (returned in number of milliseconds) and addition/subtraction of an integer (number of milliseconds).
11.1.4 The type datetime

The type datetime is used to represent a timestamp by combining a date and a time. A datetime object can be initialized by a text string, a pair date and time or a numerical value. In the first case, the conversion is processed using a predefined time format (see datetimefmt); in the third case, the number is interpreted as the number of seconds elapsed since 1/1/1970 at midnight. If the provided number is a real value, the fractional part is stored as a number of milliseconds. The constant SYS_NOW may also be used to initialize a datetime: datetime(SYS_NOW) is the current date and time. This type also supports assignment, comparison as well as difference (returned in number of seconds) and addition/subtraction of a numerical value (number of seconds).

11.2 Control parameters

Via the getparam function and the setparam procedure it is possible to access the following control parameters of module mmsystem (the reader is reminded that parameters may be spelled with lower or upper case letters or a mix of both):

- datefmt
  - Date text format.
  - p. 199
- datetimefmt
  - Date and time text format.
  - p. 200
- monthnames
  - List of month names.
  - p. 200
- sys_endparse
  - End of parsing position.
  - p. 201
- sys_fillchar
  - Padding character for text resize.
  - p. 201
- timefmt
  - Time text format.
  - p. 200

### datefmt

**Description**
Define the text format for both reading and writing a date.

**Type**
String, read/write

**Default value**
"%.y-%0m-%0d"

**Note**
The date format consists in a text string in which the date information (like day number) is specified using tags. A tag begins by the character "%" optionally followed by "." or ".0" and a character indicating which specific information must be provided. The possible values are:
- C Century
- Y Year number in the century
- y Year
- m Month (1-12)
- N Name of month according to parameter monthnames
- d Day (1-31)

The symbol "%" is optionally used, the corresponding information is produced in fixed format with space (" ") or zero ("0") as the padding character. For instance, the day 1 will be displayed as "1" with the format "%d"; as " 1" with ".d" and as "01" with "%0d".

**See also**
datetimefmt, monthnames

mmsystem 199 Mosel Reference Manual
### timefmt

**Description**  
Define the text format for both reading and writing time.

**Type**  
String, read/write

**Default value**  
"%0H:%0M:%0S,%s"

**Note**  
The time format consists in a text string in which the time information (like number of seconds) is specified using tags. A tag begins by the character "%" optionally followed by "." or "0" and a character indicating which specific information must be provided. The possible values are:
- **H** Hour (0-23)
- **h** Hour (1-12)
- **M** Minute (0-59)
- **S** Seconds (0-59)
- **s** Milliseconds (0-999)
- **p** text "pm" or "am"
- **P** text "PM" or "AM"
- **%** The symbol "%"

If the second character is used, the corresponding information is produced in fixed format with space (“.”) or zero (“0”) as the padding character. For instance, the hour 1 will be displayed as "1" with the format "%H"; as " 1" with "%.H" and as "01" with "%0H".

**See also**  
datetimefmt

### datetimefmt

**Description**  
Define the text format for both reading and writing a datetime object.

**Type**  
String, read/write

**Default value**  
"%.y-%0m-%0dT%0H:%0M:%0S,%s"

**Note**  
The datetime format accepts the syntaxes of the date formant and the time format in the same string.

**See also**  
datefmt, timefmt

### monthnames

**Description**  
Define month names to be used with the %N format.

**Type**  
String, read/write

**Default value**  
"jan feb mar apr may jun jul aug sep oct nov dec"

**Note**  
This parameter is used when converting dates from/to strings with the %N format. The string must contain 12 words separated by spaces. For conversions from strings, the comparison is not case sensitive.

**See also**  
datefmt, datetimefmt
sys_endparse

Description  Index in the text string where the parsing stopped. This parameter is updated by the 
parse* routines.
Type  Integer, read only
Set by routines  parseint, parsereal

sys_fillchar

Description  Character code used to fill empty regions generated in text strings when using the 
function setchar.
Type  Integer, read/write
Default value  32 (space character)
Affects routines  setchar

11.3  Procedures and functions

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

copytext  Copy a part of a text or string.  p. 242

cuttext  Cut a part of a text returning a copy of the deleted string.  p. 243
delttext  Delete a part of a text.  p. 244
fcopy  Copy a file.  p. 203
fdelete  Delete a file.  p. 204
findtext  Search for a string in a text or string.  p. 247
fmove  Rename or move a file.  p. 205
getasnumber  Convert a date, time or datetime into a number.  p. 229
getchar  Get a character in a string or text.  p. 248
getcwd  Get the current working directory.  p. 206
getdate  Get the date part of a datetime.  p. 214
getday  Get the day number of a date or datetime.  p. 221
getenv  Get the value of an environment variable.  p. 207
getfsize  Get the size of a file.  p. 210
getfstat  Get the status of a file or directory.  p. 209
getftime  Get time information of a file.  p. 211
gethour  Get the hour part of a time or datetime.  p. 225
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>getminute</td>
<td>Get the minute part of a time or datetime.</td>
<td>226</td>
</tr>
<tr>
<td>getmonth</td>
<td>Get the month number of a date or datetime.</td>
<td>222</td>
</tr>
<tr>
<td>getmsec</td>
<td>Get the millisecond part of a time or datetime.</td>
<td>228</td>
</tr>
<tr>
<td>getsecond</td>
<td>Get the second part of a time or datetime.</td>
<td>227</td>
</tr>
<tr>
<td>getsize</td>
<td>Get the size of a text.</td>
<td>241</td>
</tr>
<tr>
<td>getsysstat</td>
<td>Get the system status.</td>
<td>212</td>
</tr>
<tr>
<td>gettime</td>
<td>Get a time measure or the time part of a datetime.</td>
<td>213</td>
</tr>
<tr>
<td>getweekday</td>
<td>Compute the day of the week for a date or datetime.</td>
<td>224</td>
</tr>
<tr>
<td>getyear</td>
<td>Get the year part of a date or datetime.</td>
<td>223</td>
</tr>
<tr>
<td>inserttext</td>
<td>Paste a text or string into a text.</td>
<td>246</td>
</tr>
<tr>
<td>isvalid</td>
<td>Check whether a date, time or datetime is valid.</td>
<td>230</td>
</tr>
<tr>
<td>makedir</td>
<td>Create a new directory in the given file system.</td>
<td>215</td>
</tr>
<tr>
<td>openpipe</td>
<td>Start an external process for bidirectional communication.</td>
<td>216</td>
</tr>
<tr>
<td>parseextn</td>
<td>Initialise an object of a module type from a text.</td>
<td>253</td>
</tr>
<tr>
<td>parseint</td>
<td>Convert a text into an integer.</td>
<td>251</td>
</tr>
<tr>
<td>parsereal</td>
<td>Convert a text into a real.</td>
<td>252</td>
</tr>
<tr>
<td>pastetext</td>
<td>Paste a text or string into a text.</td>
<td>245</td>
</tr>
<tr>
<td>qsort</td>
<td>Sort a list or an array or (a subset of) the indices of an array.</td>
<td>217</td>
</tr>
<tr>
<td>readtextline</td>
<td>Read a line of text from the current input stream.</td>
<td>250</td>
</tr>
<tr>
<td>removedir</td>
<td>Remove a directory.</td>
<td>218</td>
</tr>
<tr>
<td>setchar</td>
<td>Set a character in a text.</td>
<td>249</td>
</tr>
<tr>
<td>setdate</td>
<td>Set the date part of a datetime.</td>
<td>239</td>
</tr>
<tr>
<td>setday</td>
<td>Set the day number of a date or datetime.</td>
<td>231</td>
</tr>
<tr>
<td>setenv</td>
<td>Set the value of an environment variable.</td>
<td>208</td>
</tr>
<tr>
<td>sethour</td>
<td>Set the hour part of a time or datetime.</td>
<td>234</td>
</tr>
<tr>
<td>setminute</td>
<td>Set the minute part of a time or datetime.</td>
<td>235</td>
</tr>
<tr>
<td>setmonth</td>
<td>Set the month number of a date or datetime.</td>
<td>232</td>
</tr>
<tr>
<td>setmsec</td>
<td>Set the millisecond part of a time or datetime.</td>
<td>237</td>
</tr>
<tr>
<td>setsecond</td>
<td>Set the second part of a time or datetime.</td>
<td>236</td>
</tr>
<tr>
<td>settime</td>
<td>Set the time part of a datetime.</td>
<td>238</td>
</tr>
<tr>
<td>setyear</td>
<td>Set the year part of a date or datetime.</td>
<td>233</td>
</tr>
<tr>
<td>sleep</td>
<td>Suspend execution for a fixed amount of time.</td>
<td>219</td>
</tr>
<tr>
<td>system</td>
<td>Execute an external program.</td>
<td>220</td>
</tr>
<tr>
<td>textfmt</td>
<td>Create a formatted text from a string, a text or a number.</td>
<td>240</td>
</tr>
</tbody>
</table>
Purpose
Copy a file.

Synopsis
procedure fcopy(namesrc:string, namedest:string)

Arguments

  namesrc  The name of the file to be copied
  namedest The destination name

Further information
This procedure copies the file namesrc to namedest. The provided names may use extended notation.
fdelete

Purpose
Delete a file.

Synopsis
procedure fdelete(filename:string)

Argument
filename   The extended name of the file to be deleted

Further information
The provided name may use extended notation.

Related topics
removedir.
fmove

Purpose
Rename or move a file.

Synopsis
procedure fmove(namesrc:string,namedest:string)

Arguments
namesrc   The name 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. The provided names may use extended notation.
**getcwd**

**Purpose**
Get the current working directory.

**Synopsis**
```
function getcwd: string
```

**Return value**
The 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

Argument
name Name of the environment variable

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

Further information
This procedure is included in the published interface of mmsystem (see Section 11.5).

Example
The value of the environment variable PATH is retrieved as follows:

        str:= getenv("PATH")

Related topics
setenv
setenv

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

**Synopsis**
procedure setenv(name:string,value:string)

**Arguments**
- **name**  Name of the environment variable
- **value**  New value for the environment variable

**Further information**
1. The environment variable is deleted if it is assigned an empty string.
2. Variables created or modified with this procedure can be retrieved using the `getenv` function and are inherited by processes started by `system` or `openpipe`.
3. The effect of this procedure is local to the running model (*i.e.* system calls like the C function `getenv` will not work for these variables). However, another module may access the environment maintained by `mmsystem` using the IMCI function `getenv` (see Section 11.5).
4. This procedure is included in the published interface of `mmsystem` (see Section 11.5).

**Related topics**
- `getenv`, `system`, `openpipe`
getfstat

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

Synopsis
function getfstat(filename:string):integer

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

Return value
Bit encoded type and mode of the given file.

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

   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 returned status type 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
getfsize

Purpose
Get the size of a file.

Synopsis
function getfsize(filename:string):integer

Argument
filename  Name (and path) of the file

Return value
The size of the file in bytes or -1 in case of error

Further information
The function returns -1 if the file cannot be found or accessed and INT_MAX if the size exceeds the integer capacity (~2Gb).
**getftime**

**Purpose**
Get time information of a file.

**Synopsis**

```plaintext
function getftime(filename:string,what:integer):real
```

**Arguments**

- **filename** Name (and path) of the file
- **what** Information requested. Possible values:
  - `SYS_FTIM_ACC` Last access
  - `SYS_FTIM_MOD` Last modification

**Return value**

The time requested as the number of seconds elapsed since 1/1/1970 at midnight or 0 in case of error.
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 or the time part of a datetime.

**Synopsis**

```plaintext
function gettime:real
function gettime(dt:datetime):time
```

**Argument**

- `dt` A datetime object

**Return value**

Time measure in seconds or a time object.

**Example**

The following prints the program execution time:

```plaintext
starttime := gettime ! Get the start time
... ! Do something
write("Time: ", gettime-starttime)
```

**Further information**

1. The measure returned by this function corresponds to the elapsed time since the module has been initialized (just before execution of the model starts).

2. The second form of this function is used to extract the time part of a datetime structure.

**Related topics**

- `getdate`
getdate

**Purpose**
Get the date part of a datetime.

**Synopsis**
```plaintext
function getdate(dt:datetime):date
```

**Argument**
dt   A datetime object

**Return value**
A date object.

**Related topics**
-gettime
**makedir**

**Purpose**
Create a new directory in the given file system.

**Synopsis**
```
procedure makedir(dirname:string)
```

**Argument**
- **dirname** The name and path of the directory to be created

**Related topics**
- `removedir`
openpipe

**Purpose**
Start an external process for bidirectional communication.

**Synopsis**
procedure openpipe(cmd:string)

**Argument**
cmd The command to be executed in the separate process

**Example**
The following example uses an external program sort (we assume it writes a sorted copy of what it reads) to display a sorted list of the content of set ToSort:

```mosel
openpipe("sort")
forall(i in ToSort)
  writeln(i)
fclose(F_OUTPUT)

while(not iseof) do
  readln(l)
  writeln(l)
end-do
fclose(F_INPUT)
```

**Further information**

1. Pipes required by this procedure are created using the pipe driver of this module (see Section 11.4.1). As a consequence, the string provided as argument must be suitable for the driver (i.e. a program name followed by its options separated by spaces).

2. This procedure opens both an input and output streams that must be closed explicitly using fclose. Note that the output stream must be closed first otherwise the program may lock up.
qsort

Purpose
Sort a list or an array or (a subset of) the indices of an array.

Synopsis
procedure qsort(sense:boolean, lvals:list of integer)
procedure qsort(sense:boolean, lvals:list of real)
procedure qsort(sense:boolean, lvals:list of string)
procedure qsort(sense:boolean, vals:array of integer)
procedure qsort(sense:boolean, vals:array of real)
procedure qsort(sense:boolean, vals:array of string)
procedure qsort(sense:boolean, vals:array of integer, ndx:array)
procedure qsort(sense:boolean, vals:array of real, ndx:array)
procedure qsort(sense:boolean, vals:array of string, ndx:array)
procedure qsort(sense:boolean, vals:array of integer, ndx:array, sel:set)
procedure qsort(sense:boolean, vals:array of real, ndx:array, sel:set)
procedure qsort(sense:boolean, vals:array of string, ndx:array, sel:set)

Arguments
sense  Sense of the sorting:
      SYS_UP   Ascending order
      SYS_DOWN  Descending order
lvals  List to be sorted
vals   One-dimensional array to be sorted
ndx    One-dimensional array of the same type and size as the indexing set of vals
sel    Subset of the indexing set of vals

Example
The following example sorts an array of real numbers:

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

ar:: [1.2, -3, -8, 10.5, 4, 7, 2.9, -1, 0, 5]
qsort(true, ar)
writeln("Sorted array: ", ar)

Further information
1. In the first three versions of the procedure (with two arguments, sense and vals) the input array vals is overwritten by the resulting sorted array.
2. When an index set ndx is provided, the resulting sorted array is returned in the argument ndx in the form of its sorted index set. If a selection set sel of indices is provided, only the specified indices are processed.
removedir

**Purpose**
Remove a directory.

**Synopsis**

```plaintext
procedure removedir(dirmame:string)
```

**Argument**

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.
sleep

**Purpose**
Suspend execution for a fixed amount of time.

**Synopsis**
procedure sleep(duration:int)

**Argument**
duration  Sleep time in milliseconds

**Further information**
The model uses no CPU while it is suspended.
**Purpose**

Execute an external program.

**Synopsis**

```plaintext
procedure system(command: string)
```

**Argument**

- `command`: The command to be executed

**Example**

The following displays the functionality of the `mmsystem` module using the program `mosel`:

```plaintext
system('mosel -s -c "exam mmsystem"
```

**Further information**

1. The given program is executed directly: if the specified expression is a shell command, it is necessary to call the shell explicitly. For instance to get a directory listing under Windows the command will be "cmd /C dir".

2. Using 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.

3. The generated process inherits the current system environment plus the environment variables modified/created using the `setenv` procedure.

4. The default output and error streams of the generated process are redirected to the corresponding Mosel streams. The default input stream is closed. Note that under Windows, the redirection of streams requires the external program `mmredir.exe`.

5. This procedure is included in the published interface of `mmsystem` (see Section 11.5).
getday

Purpose
Get the day number of a date or datetime.

Synopsis
function getday(d:date):integer
function getday(dt:datetime):integer

Arguments
d A date object
dt A datetime object

Return value
Day number in the month.

Related topics
getyear, getmonth
getmonth

Purpose
Get the month number of a date or datetime.

Synopsis
function getmonth(d:date):integer
function getmonth(dt:datetime):integer

Arguments
- d  A date object
- dt A datetime object

Return value
Month number in the year.

Related topics
getyear, getday
getyear

**Purpose**
Get the year part of a date or datetime.

**Synopsis**
```
function getyear(d:date):integer
function getyear(dt:datetime):integer
```

**Arguments**
- d A [date object](#)
- dt A [datetime object](#)

**Return value**
Year as an integer.

**Related topics**
- [getmonth](#), [getday](#)
getweekday

Purpose
Compute the day of the week for a date or datetime.

Synopsis

function getweekday(d:date):integer
function getweekday(dt:datetime):integer

Arguments
d A date object
dt A datetime object

Return value
The number of the day in the week (1-7).

Further information
The first day of the week (number 1) is Monday.
gethour

**Purpose**
Get the hour part of a time or datetime.

**Synopsis**

```plaintext
function gethour(t:time):integer
function gethour(dt:datetime):integer
```

**Arguments**
- `t` A time object
- `dt` A datetime object

**Return value**
Hour as an integer.

**Related topics**
- `getminute`
- `getsecond`
- `getmsec`
getminute

Purpose
Get the minute part of a time or datetime.

Synopsis
function getminute(t:time):integer
function getminute(dt:datetime):integer

Arguments
| t | A time object |
| dt | A datetime object |

Return value
Minute as an integer.

Related topics
gethour, getsecond, getmsec
getsecond

**Purpose**
Get the second part of a time or datetime.

**Synopsis**
```plaintext
function getsecond(t:time):integer
function getsecond(dt:datetime):integer
```

**Arguments**
- `t`: A time object
- `dt`: A datetime object

**Return value**
Second as an integer.

**Related topics**
- gethour, getminute, getmsec
**Purpose**
Get the millisecond part of a time or datetime.

**Synopsis**
```
function getmsec(t:time):integer
function getmsec(dt:datetime):integer
```

**Arguments**
- `t` A time object
- `dt` A datetime object

**Return value**
Millisecond as an integer.

**Related topics**
- `gethour`, `getminute`, `getsecond`
getasnumber

Purpose
Convert a date, time or datetime into a number.

Synopsis
function getasnumber(d:date):integer
function getasnumber(t:time):integer
function getasnumber(dt:datetime):real

Arguments
  d    A date object
  t    A time object
  dt   A datetime object

Return value
The numerical representation of the argument.

Further information
A date is converted to an integer Julian Day Number (number of days since 1/1/1970 at midnight). This function returns an integer number of milliseconds for a time and a real number of seconds for a datetime. This number represents the number of seconds and milliseconds (as the fractional part of the number) since 1/1/1970 at midnight.
isvalid

**Purpose**
Check whether a date, time or datetime is valid.

**Synopsis**
function isvalid(d:date):boolean  
function isvalid(t:time):boolean  
function isvalid(dt:datetime):boolean

**Arguments**
- **d**: A date object  
- **t**: A time object  
- **dt**: A datetime object

**Return value**
True if the argument is valid.

**Further information**
A date is valid if its month number is in the range 1-12 and its day number is in the range 1-31 and is compatible with its month number (for instance 2006-2-29 is not a valid date). A time is valid if it is positive and smaller than an entire day. A datetime is valid if both its date part and its time part are valid.
setday

**Purpose**
Set the day number of a date or datetime.

**Synopsis**
```plaintext
procedure setday(d:date, j:integer)
procedure setday(dt:datetime, j:integer)
```

**Arguments**
- **d**  A date object
- **dt** A datetime object
- **j**   Day number

**Related topics**
- setyear, setmonth
setmonth

**Purpose**
Set the month number of a date or datetime.

**Synopsis**

```plaintext
procedure setmonth(d:date,m:integer)
procedure setmonth(dt:datetime,m:integer)
```

**Arguments**

- `d` A date object
- `dt` A datetime object
- `m` Month number

**Related topics**

- `setyear`, `setday`
setyear

Purpose
Set the year part of a date or datetime.

Synopsis
procedure setyear(d:date,y:integer)
procedure setyear(dt:datetime,y:integer)

Arguments
d     A date object
dt    A datetime object
y     Year

Related topics
setmonth, setday
sethour

**Purpose**
Set the hour part of a time or datetime.

**Synopsis**

```plaintext
procedure sethour(t:time,h:integer)
procedure sethour(dt:datetime,h:integer)
```

**Arguments**

- **t** A time object
- **dt** A datetime object
- **h** Hour

**Related topics**

- setminute, setsecond, setmsec
setminute

Purpose
Set the minute part of a time or datetime.

Synopsis
procedure setminute(t:time,m:integer)
procedure setminute(dt:datetime,m:integer)

Arguments

  t  A time object
  dt A datetime object
  m  Minute

Related topics
sethour, setsecond, setmsec
setsecond

**Purpose**
Set the second part of a time or datetime.

**Synopsis**

```plaintext
procedure setsecond(t:time, s:integer)
procedure setsecond(dt:datetime, s:integer)
```

**Arguments**

- `t`: A `time` object
- `dt`: A `datetime` object
- `s`: Second

**Related topics**

- `sethour`
- `setminute`
- `setmsec`
setmsec

**Purpose**
Set the millisecond part of a time or datetime.

**Synopsis**
```plaintext
procedure setmsec(t:time,ms:integer)
procedure setmsec(dt:datetime,ms:integer)
```

**Arguments**
- **t** A time object
- **dt** A datetime object
- **ms** Millisecond

**Related topics**
- sethour, setminute, setsecond
**settime**

**Purpose**
Set the time part of a datetime.

**Synopsis**
```plaintext
procedure settime(dt:datetime,t:time)
```

**Arguments**
- `dt` A datetime object
- `t` A time object

**Related topics**
- `setdate`


setdate

**Purpose**
Set the date part of a datetime.

**Synopsis**

```plaintext
procedure setdate(dt:datetime, d:date)
```

**Arguments**

- **dt**: A **datetime** object
- **d**: A **date** object

**Related topics**

- settime
textfmt

Purpose
Create a formatted text from a string, a text or a number.

Synopsis
function textfmt (str:string, len:integer):text
function textfmt (txt:text, len:integer):text
function textfmt (i:integer, len:integer):text
function textfmt (r:real, len:integer):text
function textfmt (r:real, len:integer, dec:integer):text

Arguments
str   String to be formatted
txt   Text 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 this 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 text.

Example
The following:

    writeln("text1", textfmt("text2",8), "text3")
    writeln("text1", textfmt("text2",-8), "text3")
    r:=789.123456
    writeln(textfmt(r,0)," ", textfmt(r,4,2), textfmt(r,8,0))

produces this output:

    text1 text2 text3
    text1 text2 text3
    789.123 789.12 789

Further information
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.
getsiz

**Purpose**
Get the size of a text.

**Synopsis**
function getsize(txt:text):integer

**Argument**
txt A text object

**Return value**
The number of characters included in the text.
copytext

Purpose
Copy a part of a text or string.

Synopsis
function copytext(txt:text, i1:integer, i2:integer):text
function copytext(str:string, i1:integer, i2:integer):text

Arguments
txt  A text object
str  String
i1   Starting position of the region to copy
i2   End position of the region to copy

Return value
A copy of the region.

Example
The following:
   writeln(copytext("abcdefgh",3,7))
   writeln(copytext("abcdefgh",7,10))

produces this output:
   cdefg
   gh

Further information
This function returns an empty text 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).

Related topics
deltext, inserttext, pastetext, cuttext
**cuttext**

**Purpose**
Cut a part of a text returning a copy of the deleted string.

**Synopsis**

```plaintext
function cuttext(txt:text, i1:integer, i2:integer):text
```

**Arguments**
- `txt`: A text object
- `i1`: Starting position of the region to cut
- `i2`: End position of the region to cut

**Return value**
A copy of the region. The input text is modified accordingly.

**Example**
The following:

```plaintext
t:=text("abcdefgh")
writeln(cuttext(t,3,7))
writeln(t)
```

produces this output:

```plaintext
cdefg
abh
```

**Further information**
This function returns an empty text 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).

**Related topics**
- `deltext`
- `inserttext`
- `pastetext`
- `copytext`
**Purpose**
Delete a part of a text.

**Synopsis**
procedure deltext(txt:text, i1:integer, i2:integer)

**Arguments**
- txt A text object
- i1 Starting position of the region to delete
- i2 End position of the region to delete

**Example**
The following:
```
t:=text("abcdefgh")
deltext(t,3,7)
writeln(t)
```
produces this output:
```
abh
```

**Related topics**
cuttext, inserttext, pastetext, copytext
**pastetext**

**Purpose**
Paste a text or string into a text.

**Synopsis**

```plaintext
procedure pastetext(txt:text, str:string, start:integer)
procedure pastetext(txt:text, src:text, start:integer)
```

**Arguments**
- `txt` A text object
- `src` A text object
- `str` A string
- `start` Paste position

**Example**
The following:

```plaintext
t:=text("abcdefgh")
pastetext(t,"123",2)
writeln(t)
pastetext(t,"456",8)
writeln(t)
```

produces this output:

```
a123efgh
a123efg456
```

**Related topics**
- `cuttext`
- `inserttext`
- `deltext`
- `copytext`
**inserttext**

**Purpose**
Paste a text or string into a text.

**Synopsis**

```plaintext
procedure inserttext(txt:text, str:string, start:integer)
procedure inserttext(txt:text, src:text, start:integer)
```

**Arguments**

- **txt**  A text object
- **src**  A text object
- **str**  A string
- **start**  Insert position

**Example**

The following:

```plaintext
t:=text("abcdefgh")
inserttext(t,"123",2)
writeln(t)
inserttext(t,"456",8)
writeln(t)
```

produces this output:

```
a123bcdefgh
a123bcd456efgh
```

**Related topics**

- cuttext, deltext, pastetext, copytext
findtext

Purpose
Search for a string in a text or string.

Synopsis
function findtext(txt: text, toft: text, start: integer): integer
function findtext(txt: text, tofs: string, start: integer): integer
function findtext(str: string, tofs: string, start: integer): integer

Arguments
- txt: A text object
- str: String
- toft: Text to find
- tofs: String to find
- start: Starting position for the search

Return value
Index of the string or 0 if not found.

Example
The following:

writeln(findtext("abcdefgh","de",2))
writeln(findtext("abcdefgh","de",5))

produces this output:

4
0
**getchar**

**Purpose**
Get a character in a string or text.

**Synopsis**
function getchar(txt:text, index:integer):integer
function findtext(str:string, index:integer):integer

**Arguments**
- **txt** A text object
- **str** String
- **index** Position of the character

**Return value**
Character code or -1 if the index is not valid.

**Related topics**
setchar
setchar

Purpose
Set a character in a text.

Synopsis
procedure setchar(txt:text, index:integer, c:integer)

Arguments
<table>
<thead>
<tr>
<th>txt</th>
<th>A text object</th>
</tr>
</thead>
<tbody>
<tr>
<td>str</td>
<td>String</td>
</tr>
<tr>
<td>index</td>
<td>Position of the character</td>
</tr>
<tr>
<td>c</td>
<td>Character code</td>
</tr>
</tbody>
</table>

Further information
If the index requested is after the end of the text, the text is expended as necessary and the newly created space is padded with the character which code is the parameter `sys_fillchar`.

Related topics
getchar, sys_fillchar
readtextline

**Purpose**
Read a line of text from the current input stream.

**Synopsis**

```plaintext
function readtextline(txt: text): integer
```

**Argument**

<table>
<thead>
<tr>
<th>txt</th>
</tr>
</thead>
<tbody>
<tr>
<td>A text object</td>
</tr>
</tbody>
</table>

**Return value**

Number of characters read or -1 if end of file.
parseint

Purpose
Convert a text into an integer.

Synopsis
function parseint(txt:text,start:integer):integer
function parseint(txt:text,start:integer,base:integer):integer

Arguments
- **txt**: A text object
- **start**: Starting position in the text
- **base**: Base to use for the conversion

Return value
The integer represented by the string.

Example
The following:
```mosel
  t:=text("a123.4b")
  writeln(parseint(t,2))
  writeln(getparam("sys_endparse"))
```
produces this output:
```
  123
  5
```

Further information
1. The parsing begins at the specified starting position and stops as soon as an invalid character is found. This location is then stored in the parameter `sys_endparse`.

2. The optional **base** argument may be used if the text is not expressed in base 10. Valid values for this parameter is 0 and 2 to 36. If base is zero or 16, the string may then include a ‘0x’ prefix, and the number will be read in base 16. Furthermore, if the base is 0, the text will be read in base 8 if the first character is 0 and in base 10 otherwise.

Related topics
- `parsereal`, `parseextn`, `sys_endparse`
parsereal

Purpose
Convert a text into a real.

Synopsis
function parsereal(txt:text, start:integer):real

Arguments
- txt: A text object
- start: Starting position in the text

Return value
The real represented by the string.

Example
The following:

```mosel
  t:=text("a123.4b")
  writeln(parsereal(t,2))
  writeln(getparam("sys_endparse"))
```

produces this output:

```
123.4
7
```

Further information
The parsing begins at the specified starting position and stops as soon as an invalid character is found. This location is then stored in the parameter sys_endparse.

Related topics
parseint, parseextn, sys_endparse
**parseextn**

**Purpose**
Initialise an object of a module type from a text.

**Synopsis**
```
procedure(txt: text, start: integer, e: mtype)
```

**Arguments**
- `txt`: A `text` object
- `start`: Starting position in the text
- `e`: An object of an external type

**Further information**
This function can only be used with types supporting initialisation from a string.

**Related topics**
- `parseint`, `parsereal`
11.4 I/O drivers

The mmsystem module provides an IO driver to connect a Mosel input or output stream to a program started in a different process. Using this driver, it is possible to get the output of an external program (for instance the result of a preprocessor to feed the Mosel compiler) or implement a basic bidirectional inter process communication thanks to the openpipe procedure (which relies on this IO driver).

11.4.1 Driver pipe

pipe:program [options...]

The file name for this driver is an external program with its options. Options are separated by spaces or tabulations and may be quoted using either single or double quotes. A quoted option may contain any kind of character except the quote used to delimit the string.

When the system opens a pipe, a new process is started for executing the given program and default input and output streams are directed to system pipes. If the file is open for reading (resp. writing), the default output stream (resp. input stream) of the new process becomes the current input stream (resp. output stream) of the model. To locate the program to be executed, the system relies on the PATH environment variable. Detection of error (typically the program cannot be found or is not executable) differs depending on the operating system: under Windows, the error is reported immediately and the pipe is not open. With Posix systems, no error is reported but following IO operations fail.

When the file is closed, both input and output streams of the external process are closed then the system waits for its termination: in order to avoid a lock up of the Mosel program one must make sure that the external program ends its execution when default input and output streams are closed.

Example: the following command could be used with Mosel Console for compiling the model mymod.mos after it has been processed by the C preprocessor. Note that we have to provide an output file name since the compiler cannot deduce it from the source file name.

For a Posix systems:

```
compile 'mmsystem.pipe:cpp mymod.mos' '' mymod.bim
```

For Windows (with MSVC):

```
compile 'mmsystem.pipe:cl /E mymod.mos' '' mymod.bim
```

11.5 Published library functions

The module mmsystem publishes its implementation of getenv, setenv and system as well as the functions gettxtsize, gettxtbuf and txtresize for text access 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 mmsystem_imci that is defined in the module header file mmsystem.h.

From another module, the context of mmsystem 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 dso;
mmsystem_imci mmsys;
void *sysctx;

dso=mm->finddso("mmsystem"); // Retrieve the mmsystem module*/```
sysctx=*(mm->getdsoctx(mmctx, dso, {void *}{&mmsys}));
/* Get the module context and the communication interface of mmsystem */

Typically, a module calling functions that are provided by mmsystem will include this module into its list of dependencies in order to make sure that mmsystem 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:

    static const char *deplist[]={"mmsystem", NULL}; /* Module dependency list */
    static XPRMdsoserv tabserv[] =
    {
        {XPRM_SRV_DEPLST, (void *)deplist}
    };

Using these functions a module may access and modify the environment of the calling model and execute an external program with automatic redirection of default streams:

    mmsys->setenv(ctx, sysctx, "MYVAR", "A_VALUE");
    rts=mmsys->system(ctx, sysctx, "myprogram arg1 arg2");

11.5.1 Description of the library functions

gettxtbuf  Get a reference to the character buffer of a text object.  p. 257
gettxtsize Get the size of a text object.  p. 256
txtresize  Resize a text object.  p. 258
gettxtsize

Purpose
Get the size of a text object.

Synopsis
int gettxtsize(XPRMctx ctx, void *sysctx, void *t);

Arguments
ctx  Mosel's execution context
sysctx  Context of mmsystem
t  Reference to a text object

Return value
The size of the character buffer (excluding the terminating 0 character).

Related topics
txtresize, gettxtbuf
gettxtbuf

Purpose
Get a reference to the character buffer of a text object.

Synopsis
char *gettxtbuf(XPRMctx ctx, void *sysctx, void *t);

Arguments
ctx Mosel’s execution context
sysctx Context of mmsystem
t Reference to a text object

Return value
A reference to the character buffer.

Further information
1. The buffer returned is terminated by the character 0 (like a C string) and can be modified as long as the size is not changed. If the length of the buffer has to be altered, use the function txtresize.

2. Since the memory management of the module may move text buffers when allocating memory, the pointer returned by this function is only valid until the next memory allocation.

Related topics
txtresize, gettxtsize
**Purpose**
Resize and get a reference to the character buffer of a text object.

**Synopsis**
```c
char *txtresize(XPRMctx ctx, void *sysctx, void *t, int s);
```

**Arguments**
- **ctx**: Mosel’s execution context
- **sysctx**: Context of `mmsystem`
- **t**: Reference to a text object
- **s**: New size of the buffer (terminating 0 is not counted)

**Return value**
A reference to the new character buffer.

**Further information**
1. The buffer returned is terminated by the character 0 (like a C string) and can be modified as long as the size is not changed.
2. Since the memory management of the module may move text buffers when allocating memory, the pointer returned by this function is only valid until the next memory allocation.

**Related topics**
- `gettxtsize`
Chapter 12

mmxprs

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

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 manual, further details relating to the usage of these may be found by consulting the Xpress-Optimizer Reference Manual.

12.1 New functionality for the Mosel language

12.1.1 The type basis

The module mmxprs defines the type basis to represent solution basis in the Mosel Language. This new type is used to store a basis computed by the optimizer during its solution process (savebasis). A basis can then be loaded again into the optimiser with loadbasis, inspected (by getting the basis status of each variable/constraint it includes with getbstatus) or modified (by changing this basis status using setbstat). The type basis supports assignment and test of equality. This comparison only checks whether two basis contain the same information, it does not indicate whether the basis are equivalent.

12.2 Control parameters

This module extends the getparam function and the setparam procedure in order to access all the control and problem parameters of 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>Parameter</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>XPRS_colorder</td>
<td>Reorder matrix columns before loading the problem.</td>
<td>260</td>
</tr>
<tr>
<td>XPRS_loadnames</td>
<td>Enable/disable loading of MPS names into the Optimizer.</td>
<td>260</td>
</tr>
<tr>
<td>XPRS_problem</td>
<td>Optimizer problem pointer</td>
<td>260</td>
</tr>
<tr>
<td>XPRS_probname</td>
<td>Read/set the problem name used by the Optimizer.</td>
<td>260</td>
</tr>
<tr>
<td>XPRSVerbose</td>
<td>Enable/disable message printing by the Optimizer</td>
<td>260</td>
</tr>
</tbody>
</table>

Example:

```
setparam("XPRS_verbose", true) ! Turn on message printing
pstat:= getparam("XPRS_lpstatus") ! Get the problem LP optimization status
```
XPRS_colorder

**Description**  Reorder matrix columns before loading the problem.

**Type**  Integer, read/write

**Values**
- 0  Mosel implicit ordering
- 1,3  Reorder using a numeric criterion
- 2  Alphabetical order of the variable names (this requires the names to be available)

**Default value**  0

XPRS_loadnames

**Description**  Enable/disable loading of MPS names into the Optimizer.

**Type**  Boolean, read/write

**Values**
- true  Enable loading of names
- false  Disable loading of names

**Default value**  false

**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**  String, read only

XPRS_probnname

**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 operating system is generated.

**Type**  String, read/write

XPRS_verbose

**Description**  Enable/disable message printing by the Optimizer

**Type**  Boolean, read/write
12.3 Procedures and functions

This section lists in alphabetical order the functions and procedures that are provided by the `mmxprs` module.

- **clearmipdir** Delete all defined MIP directives. p. 263
- **clearmodcut** Delete all defined model cuts. p. 264
- **command** Execute an Optimizer command. p. 265
- **defsecurevecs** Define the sets of variables and constraints that must not be removed by the presolve. p. 266
- **fixglobal** Fix values of global entities. p. 268
- **getbstat** Get the status of a variable or constraint in a basis. p. 269
- **getiis** Compute then get the Irreductible Infeasible Sets (IIS). p. 270
- **getinfeas** Returns sets of infeasible primal and dual variables. p. 271
- **getlb** Get the lower bound of a variable. p. 272
- **getname** Get the name of a decision variable or constraint. p. 273
- **getprobstat** Get the Optimizer problem status. p. 274
- **getrange** Get a range value for a variable or constraint. p. 275
- **getsensrng** Get sensitivity ranges for objective or RHS function coefficients. p. 276
- **getub** Get the upper bound of a variable. p. 277
- **initglobal** Reset the global search. p. 278
- **isintegral** Check whether a solution value is integral. p. 279
- **loadbasis** Load a previously saved basis. p. 280
- **loadmipsol** Load a MIP solution into the optimizer. p. 281
- **loadprob** Load a problem into the optimizer. p. 282
- **maximize, minimize** Maximize/minimize the current problem. p. 284
- **readbasis** Read a basis from a file. p. 285
- **readdirs** Read directives from a file. p. 286
- **resetbasis** Reset a basis. p. 287
- **savebasis** Save the current basis. p. 288
- **savemipsol** Save the current solution into the provided array. p. 289
- **savestate** Save current state of the Optimizer to a file. p. 290
- **setbstat** Set the status of a variable or constraint in a basis. p. 291
- **setcallback** Set optimizer callback functions and procedures. p. 292
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setlb</code></td>
<td>Set the lower bound of a variable.</td>
<td>294</td>
</tr>
<tr>
<td><code>setmipdir</code></td>
<td>Set a directive on a variable or Special Ordered Set.</td>
<td>295</td>
</tr>
<tr>
<td><code>setmodcut</code></td>
<td>Mark a constraint as a model cut.</td>
<td>296</td>
</tr>
<tr>
<td><code>setub</code></td>
<td>Set the upper bound of a variable.</td>
<td>297</td>
</tr>
<tr>
<td><code>setucbdata</code></td>
<td>Update data for CHGBRANCH callback.</td>
<td>298</td>
</tr>
<tr>
<td><code>stopoptimize</code></td>
<td>Interrupt the optimizer algorithms.</td>
<td>267</td>
</tr>
<tr>
<td><code>unloadprob</code></td>
<td>Unload the problem held in the optimizer.</td>
<td>283</td>
</tr>
<tr>
<td><code>writebasis</code></td>
<td>Write the current basis to a file.</td>
<td>299</td>
</tr>
<tr>
<td><code>writedirs</code></td>
<td>Write current directives to a file.</td>
<td>300</td>
</tr>
<tr>
<td><code>writeprob</code></td>
<td>Write the current problem to a file.</td>
<td>301</td>
</tr>
</tbody>
</table>
clearmipdir

Purpose
Delete all defined MIP directives.

Synopsis
procedure clearmipdir

Further information
This procedure clears the list of directives defined so far.

Related topics
setmipdir.
clearmodcut

**Purpose**
Delete all defined model cuts.

**Synopsis**
procedure clearmodcut

**Further information**
This procedure clears the list of model cuts defined so far.

**Related topics**
setmodcut.
command

Purpose
Execute an Optimizer command or enter interactive mode of the Optimizer.

Synopsis

```
procedure command(cmd:string)
```

Argument

```
  cmd  Command or sequence of commands separated by "\n" character
```

Example

```
Solve a MIP problem and then enter interactive mode:

  command("minim\nglobal")
  command
```

Further information

When used without parameter, this procedure enters an interactive mode of the Optimizer
similar to the console mode: model execution is suspended and Optimizer commands can be
typed directly. Model execution resumes after command `quit` has been typed or the input
stream has reached an end of file. Using the alternate form of the procedure with an argu-
ment, one can send a command (or sequence of commands) to the Optimizer: this may be
useful to execute commands for which there is no `mmxprs` interface.

During the execution of this procedure, callbacks set up in the model are effective and the
problem solution status of `mmxprs` is updated upon termination. Note that, commands alter-
ing the problem must be avoided (like `readprob`, change of name of the problem, etc.) in
order to preserve consistency between Mosel and Optimizer representations of the problem.
defsecurevecs

Purpose
Define the sets of variables and constraints that must not be removed by the presolve.

Synopsis
procedure defsecurevecs(vset:set of mpvar,cset:set of linctr)

Arguments
vset     Set of decision variables to preserve or to reset a previous setting
cset     Set of constraints to preserve or to reset a previous setting

Further information
This procedure stores references to the provided sets that are used when the problem is loaded into the optimizer. These sets can be modified after the call to this procedure: the optimizer will use the current content of the sets at the time of loading the problem.
**stopoptimize**

**Purpose**
Interrupt the optimizer algorithms.

**Synopsis**
procedure stopoptimize(why:integer)

**Argument**

<table>
<thead>
<tr>
<th>why</th>
<th>The reason for stopping. Possible reasons:</th>
</tr>
</thead>
<tbody>
<tr>
<td>XPRS_STOP_TIMELIMIT</td>
<td>Time limit hit</td>
</tr>
<tr>
<td>XPRS_STOP_CTRLC</td>
<td>Control C hit</td>
</tr>
<tr>
<td>XPRS_STOP_NODELIMIT</td>
<td>Node limit hit</td>
</tr>
<tr>
<td>XPRS_STOP_ITERLIMIT</td>
<td>Iteration limit hit</td>
</tr>
<tr>
<td>XPRS_STOP_MIPGAP</td>
<td>MIP gap is sufficiently small</td>
</tr>
<tr>
<td>XPRS_STOP_SOLLIMIT</td>
<td>Solution limit hit</td>
</tr>
<tr>
<td>XPRS_STOP_USER</td>
<td>User interrupt</td>
</tr>
</tbody>
</table>

**Further information**
This procedure can be called from any callback. It is ignored if used from outside an optimization process.
fixglobal

**Purpose**
Fix values of global entitites according to the current solution.

**Synopsis**
procedure fixglobal

**Example**
Solve the MIP problem, reload the problem after solving, fix global entities to their solution values, and finally solve the LP for the continuous variables in order to be able to use `getrange`.

```plaintext
minimize(obj)
fixglobal
minimize(XPRS_TOP+XPRS_LIN, obj)
writeln(getrange(XPRS_UPACT,x))
```

**Further information**
This procedure fixes the non-continuous variables to their value of the current solution. A call to this function is required when performing sensitivity analysis on MIP problems using `getrange`.

**Related topics**
`getrange`.
getbstat

Purpose
Get the status of a variable or constraint in a basis.

Synopsis
function getbstat(b:basis, v:mpvar):integer
function getbstat(b:basis, c:linctr):integer

Arguments
b A basis
v A decision variable
c A linear constraint

Return value
Basis status. For a variable:
-1 Variable is not in the basis
0 Variable is non-basic at lower bound, or superbasic at zero if the variable has no lower bound
1 Variable is basic
2 Variable is non-basic at upper bound
3 Variable is super-basic

For a constraint:
-1 Constraint is not in the basis
0 Slack, surplus or artificial is non-basic at lower bound
1 Slack, surplus or artificial is basic
2 Slack or surplus is non-basic at upper bound
3 Slack or surplus is super-basic

Related topics
savebasis, setbstat, resetbasis.
getiis

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

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

**Arguments**
- `vset`  Set to return the decision variables of the IIS or if not required
- `cset`  Set to return the constraints of the IIS or if not required

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

**Related topics**
- `getinfeas`
getinfeas

**Purpose**
Returns sets of infeasible primal and dual variables.

**Synopsis**
```
procedure getinfeas(mx:set of mpvar,mslack:set of linctr,mdual:set of linctr,mdj:set of mpvar)
```

**Arguments**
- `mx` Set to return the infeasible variables or if not required
- `mslack` Set to return infeasible constraints or if not required
- `mdual` Set to return dual infeasible constraints or if not required
- `mdj` Set to return the dual infeasible variables or if not required

**Related topics**
- `getiis`
getlb

**Purpose**
Get the lower bound of a variable.

**Synopsis**
```plaintext
function getlb(x:mpvar):real
```

**Argument**
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`. 
getname

Purpose
Get the name of a decision variable or constraint of the problem.

Synopsis
function getname(x:mpvar):string
function getname(c:linctr):string

Arguments
\begin{itemize}
\item \textbf{x} A decision variable used in the problem
\item \textbf{c} A constraint (or SOS) of the problem
\end{itemize}

Return value
Name of the given object.

Further information
This function returns the name of a decision variable or constraint of the problem that would be used for matrix exportation. The parameter of this function must be part of the problem — for instance a hidden constraint cannot be assigned a name.
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
- XPRS_OTH  Unsolved or objective worse than cutoff

Example
The following procedure displays the current problem status:

```mosel
procedure print_status
declarations
    status: string
end-declarations

case getprobstat of
    XPRS_OPT: status:="Optimum found"
    XPRS_UNF: status:="Unfinished"
    XPRS_INF: status:="Infeasible"
    XPRS_UNB: status:="Unbounded"
    XPRS_OTH: status:="Failed"
    else status:="???"
end-case

writeln("Problem status: ", status)
end-procedure
```

Further information
More detailed information than what is 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).

Related topics
getparam.
**Purpose**
Get a range value for a variable or constraint.

**Synopsis**
```plaintext
function getrange(w:integer, x:mpvar):real
function getrange(w:integer, c:linctr):real
```

**Arguments**
- **w** Which information to return. Possible values:
  - XPRS_UPACT  Upper activity
  - XPRS_LOACT  Lower activity
  - XPRS_UUP    Upper unit cost
  - XPRS_UDN    Lower unit cost
  - XPRS_UCOST  Upper cost (variable only)
  - XPRS_LCOST  Lower cost (variable only)

- **x** A variable of the problem

- **c** A constraint of the problem

**Return value**
Range information depending on the value of **w**.

**Further information**
This function returns ranging information to be used for sensitivity analysis after the problem has been optimized. On MIP problems, global entities have to be “fixed” using the procedure `fixglobal` before this function can be called.

**Related topics**
`fixglobal`
getsensrng

Purpose
Get sensitivity ranges for objective or RHS function coefficients.

Synopsis
function getsensrng(w:integer, x:mpvar):real
function getsensrng(w:integer, c:linctr):real

Arguments
w Which information to return. Possible values:
XPRS_UP Upper sensitivity range
XPRS_DN Lower sensitivity range
x A variable of the problem
c A constraint of the problem

Return value
Sensitivity range information depending on the value of w.

Further information
This function returns sensitivity ranges for RHS function coefficients (if used with a constraint) and for objective function coefficients (if used with a variable). getsensrng can be called only if an optimal LP solution is available and the problem is not MIP presolved.
getub

Purpose
Get the upper bound of a variable.

Synopsis
function getub(x:mpvar):real

Argument
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.

**Synopsis**
procedure initglobal

**Further information**
This procedure resets the global search started by `maximize` or `minimize`.

**Related topics**
`maximize`, `minimize`. 
**isintegral**

**Purpose**
Check whether a variable (or set of variables) solution value is integral.

**Synopsis**
function isintegral(x:mpvar):boolean
function isintegral(s:set of mpvar):boolean

**Arguments**
- x  A decision variable
- s  A set of decision variables

**Return value**
true if the variable (or all variables of the set) is integral.

**Further information**
This function checks whether the current solution value of a variable is integral with respect to the tolerance value of the optimizer (XPRS_MIPSOL). When used with a set, the function returns true if all variables of the set satisfy the condition.
loadbasis

Purpose
Load a previously saved basis.

Synopsis
procedure loadbasis(b:basis)

Argument
b A basis

Example
The following saves a basis, changes the problem, and then loads it into the Optimizer, reloading the old basis:

declarations
  MinCost:linctr
  mybasis:basis
end-declarations

  savebasis(mybasis)
  ...
  loadprob(MinCost)
  loadbasis(mybasis)

Further information
1. This procedure loads a basis into the optimizer that has previously been saved using procedure savebasis or constructed using setbstat.

2. The problem must be loaded in the Optimzer 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.

Related topics
loadprob, savebasis, setbstat, getbstat, resetbasis.
**loadmipsol**

**Purpose**
Load a MIP solution into the optimizer.

**Synopsis**

```
function loadmipsol(s:array(set of mpvar) of real):integer
```

**Argument**

- `s`: An array containing the solution

**Return value**

Operation status:
- `-1`: Solution rejected because an error occurred
- `0`: Solution accepted
- `2`: Solution rejected because it is infeasible
- `2`: Solution rejected because it is cut off
- `3`: Solution rejected because the LP reoptimization was interrupted

**Example**

The following saves a MIP solution, modifies the problem, and then loads it into the Optimizer, reloading the MIP solution:

```mosel
declarations
MinCost:linctr
mysol: array(set of mpvar) of real
result: integer
end-declarations

savemipsol(mysol)
... ! Make some changes
loadprob(MinCost)
result:= loadmipsol(mysol)
if result<>0 then writeln("Loading MIP solution failed"); end-if
minimize(MinCost)
```

**Further information**

1. This function loads a MIP solution into the optimizer that has previously been saved using procedure `savemipsol` or constructed by some external heuristic. In the latter case a value needs to be assigned to each discrete variable in the problem, such as `mysol(x):= 1` (where `x` is a decision variable of type `mpvar`).

2. The values for the continuous variables in the `s` array are ignored and are calculated by fixing the integer variables and reoptimizing.

3. The current problem definition must be loaded into the Optimizer for `loadmipsol` to have any effect. If this has not recently been done, e.g., by calling `maximize` or `minimize`, the problem must be explicitly loaded using `loadprob`.

4. If the MIP solution is accepted by the Optimizer it causes the `MIPABSCUTOFF` control to be set accordingly. The provided MIP solution may help guiding the MIP heuristics but the branch-and-bound search will start from the initial LP relaxation solution as usual.

**Related topics**

- `savemipsol`
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 gets 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 parameter `force` can be used to force a reload of the problem in such a case. The parameter `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`.
unloadprob

**Purpose**
Unload the problem held in the optimizer.

**Synopsis**
procedure unloadprob

**Further information**
1. This procedure "unloads" the optimizer by releasing all the resources it has allocated for its processing (internal representation, solution information, working files).
2. This procedure resets the control parameters `XPRS_EXTRACOLS`, `XPRS_EXTRAROWS`, `XPRS_EXTRAEMS` to their default values.

**Related topics**
`maximize`, `minimize`, `loadprob`. 
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)

Arguments
alg    Algorithm choice:
   XPRS_BAR     Newton-Barrier to solve LP
   XPRS_DUAL    Dual simplex
   XPRS_NET     Network solver
   XPRS_LIN     Only solve LP ignoring all global entities
   XPRS_TOP     Stop after solving 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 *Profit* using the dual simplex algorithm and stops before the global search:

```mosel
declarations
    Profit:linctr
end-declarations

maximize(XPRS_DUAL+XPRS_TOP, Profit)
```

The following minimizes *MinCost* using the Newton-Barrier algorithm and ignoring all global entities:

```mosel
declarations
    MinCost:linctr
end-declarations

minimize(XPRS_BAR+XPRS_LIN, MinCost)
```

Further information
1. This procedure calls 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. But 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    Extended file name
options  String of options

Further information
This procedure reads in a basis from a file by calling the function XPRSpreadbasis 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 behavior of this procedure refer to the Xpress-Optimizer Reference Manual.

Related topics
writebasis.
readdirs

Purpose
Read directives from a file.

Synopsis
procedure readdirs(fname:string)

Argument
fname    Extended file name

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

Related topics
writedirs.
resetbasis

Purpose
Reset a basis.

Synopsis
procedure resetbasis(b:basis)

Argument
b A basis

Further information
This function clears the information stored in a basis object.

Related topics
loadbasis, savebasis, setbstat, resetbasis.
savebasis

**Purpose**
Save the current basis.

**Synopsis**

```
procedure savebasis(b:basis)
```

**Argument**

`b` A basis

**Further information**

This function saves the current basis into the provided `basis` object.

**Related topics**

`loadbasis`, `setbstat`, `getbstat`, `resetbasis`. 
savemipsol

Purpose
Save the current solution into the provided array.

Synopsis
procedure savemipsol(s:array(set of mpvar) of real)

Argument
s An array to return the solution

Further information
1. This procedure saves the current solution into the provided array. The resulting datastructure may be used as input for the loadmipsol function.

2. If the index set of the array is dynamic, the procedure may extend it in order to have all variables of the problem. Otherwise the solution is saved only for the variables included in this set.

Related topics
loadmipsol.
savestate

**Purpose**
Save current state of the Optimizer to a file.

**Synopsis**
procedure savestate(fname:string)

**Argument**
fname   Extended file name

**Further information**
The produced file can then be used as input to Optimizer console using *optimizer's* command RESTORE.
setbstat

Purpose
Set the status of a variable or constraint in a basis.

Synopsis
procedure setbstat(b:basis,v:mpvar,s:integer)
procedure setbstat(b:basis,c:linctr,s:integer)

Arguments
b    A basis
v    A decision variable
c    A linear constraint
s    Basis status. For a variable:
    -1   Remove the variable from the basis
    0    Variable is non-basic at lower bound, or superbasic at zero if the variable
         has no lower bound
    1    Variable is basic
    2    Variable is non-basic at upper bound
    3    Variable is super-basic
For a constraint:
    -1   Remove the constraint from the basis
    0    Slack, surplus or artificial is non-basic at lower bound
    1    Slack, surplus or artificial is basic
    2    Slack or surplus is non-basic at upper bound
    3    Slack or surplus is super-basic

Related topics
savebasis, getbstat, resetbasis.
**setcallback**

**Purpose**
Set optimizer callback functions and procedures.

**Synopsis**
```plaintext
procedure setcallback(cbtype:integer, cb:string)
```

**Arguments**
- **cbtype** Type of the callback:
  - XPRS_CB_LPLOG Simplex log callback
  - XPRS_CB_CUTLOG Cut 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_CHGBRANCH User choose branching variable callback
- **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
  - function cb:boolean XPRS_CB_CUTLOG
  - function cb:boolean XPRS_CB_GLOBALLOG
  - function cb:boolean XPRS_CB_BARLOG
  - function cb(node:integer):integer XPRS_CB_CHGNODE
  - function cb(node:integer):integer XPRS_CB_PRENODE
  - function cb:boolean XPRS_CB_OPTNODE
  - procedure cb XPRS_CB_INFNODE
  - procedure cb XPRS_CB_INTSOL
  - procedure cb XPRS_CB_NODECUTOFF
  - procedure cb XPRS_CB_INITCUTMGR
  - procedure cb XPRS_CB_FREECUTMGR
  - function cb:boolean XPRS_CB_CUTMGR
  - procedure cb(e:integer,u:integer,d:real) XPRS_CB_CHGBRANCH

**Example**
The following example defines a procedure to handle solution printing and sets it to be called whenever an integer solution is found using the integer solution callback:

```plaintext
procedure printsol
declarations
  objval:real
end-declarations

objval:= getparam("XPRS_lpobjval")
writeln("Solution value: ", objval)
end-procedure

setcallback(XPRS_CB_INTSOL, "printsol")
```

**Further information**
This procedure sets the optimizer callback functions and procedures. Passing an empty string (""") as the function name disables the corresponding callback. 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`. Note further that the function `setcbdata` can be used to return information to the optimizer from the callback ‘CHG-BRANCH’.
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 value 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 pseudo cost
  XPRS_PD r is a down pseudo cost

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)
```

**Argument**

`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).

**Related topics**
- `getlb`
- `getub`
- `loadprob`
- `setlb`
setucbdata

**Purpose**
Update data to be returned to the Optimizer by the CHGBRANCH callback.

**Synopsis**

```plaintext
procedure setucbdata(x:mpvar, u:integer, e:real)
procedure setucbdata(s:linctr, u:integer, e:real)
procedure setucbdata(n:integer, u:integer, e:real)
```

**Arguments**

- `x` A decision variable
- `s` An SOS
- `n` A column or SOS number as provided by the optimizer
- `u` Direction for branching. Possible values:
  - 0: Upward branch made second (branch on column)
  - 1: Upward branch made first (branch on column)
  - 2: Upward branch made second (branch on SOS)
  - 3: Upward branch made first (branch on SOS)
- `e` Estimated degradation at the node

**Further information**
This procedure stores the provided information that will be returned to the optimizer when the callback terminates. This procedure has no effect if called from outside of the CHGBRANCH callback.

**Related topics**

- `setcallback`
**writebasis**

**Purpose**
Write the current basis to a file.

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

**Arguments**
- **fname**  Extended file name
- **options**  String of options

**Further information**
This procedure writes the current basis to a file by calling the Optimizer function XPRSWritebasis. Note that basis save/read procedures can be used only if the constraint and variable names have been loaded into the Optimizer (parameter XPRS_loadnames set to true) and all constraints are named. For more detail on the options and behavior 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)

Argument
fname  Extended 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`  Extended 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 behavior of this procedure, refer to the *Xpress-Optimizer Reference Manual*.

**Related topics**
- `exportprob`
12.4 Cut Pool Manager

This section contains the functions and procedures of the Xpress-Optimizer cut manager. For a detailed description of the cut manager and its functionality the user is referred to the Xpress-Optimizer Reference Manual. To run the cut manager from Mosel, it may be necessary to (re)set certain control parameters of the optimizer. For example, switching off presolve and automatic cut generation, and reserving space for extra rows in the matrix may be useful:

```mosel
setparam("XPRS_presolve", 0); /* Switch presolve off... */
setparam("XPRS_presolveops", 2270); /* ...or use secure setting for presolve */
setparam("XPRS_cutstrategy", 0); /* Switch automatic cut generation off */
setparam("XPRS_extrarows", 5000); /* Reserve space for 5000 extra rows in the matrix*/
```

The callback functions and procedures that are relevant to the cut manager are initialized with 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.

- `addcut` Add a cut to the problem in the optimizer. p. 303
- `addcuts` Add an array of cuts to the problem in the optimizer. p. 304
- `delcuts` Delete cuts from the problem in the optimizer. p. 305
- `dropcuts` Drop a set of cuts from the cut pool. p. 306
- `getcnlist` Get the set of cuts active at the current node. p. 307
- `getcplist` Get a set of cut indices from the cut pool. p. 308
- `loadcuts` Load a set of cuts into the problem in the optimizer. p. 309
- `storecut` Store a cut into the cut pool. p. 310
- `storecuts` Store an array of cuts into the cut pool. p. 311
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 or equal)
  CT_LEQ  Inequality (less or equal)
  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, delcut.
addcuts

Purpose
Add an array of cuts to the problem in the optimizer.

Synopsis
procedure addcuts(cuttype:array(range) of integer, type:array(range) of integer, linexp:array(range) of linctr)

Arguments
- cuttype: Array of integer number for identification of the cuts
- type: Array of cut types (equation/inequality):
  - CT_GEQ: Inequality (greater or equal)
  - CT_LEQ: Inequality (less or equal)
  - 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, delcut.
delcuts

Purpose
Delete cuts from the problem in the optimizer.

Synopsis
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 non-basic 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 parameter 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**

```plaintext
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**
```plaintext
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**

```plaintext
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
  - 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 an 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**

```plaintext
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
  - `1`  Treat cut types as numbers
  - `2`  Treat cut types as bitmaps (load cut if any bit matches any bit set in `cuttype`)
  - `3`  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**
Store a cut into the cut pool.

**Synopsis**
```plaintext
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):
  - CT_GEQ: Inequality (greater or equal)
  - CT_LEQ: Inequality (less or equal)
  - CT_EQ: Equality
- **linexp** Linear expression (= unbounded constraint)

**Return value**
Index number of the cut stored in the cut pool.

**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**
- `dropcut`
- `loadcuts`
- `storecuts`
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 number for identification of the cuts
type  Array of cut types (equation/inequality):
CT_GEQ  Inequality (greater or equal)
CT_LEQ  Inequality (less or equal)
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
dropcut, loadcuts, storecut.
Appendix
Appendix A
Syntax diagrams for the Mosel language

A.1 Main structures and statements

\( \langle \text{Model} \rangle := \langle 'model' \rangle \quad \text{String} \quad \langle \text{Identifier} \rangle \quad \cdots \\
\quad \langle \text{Directives} \rangle \quad \langle \text{Parameters} \rangle \quad \langle \text{Body} \rangle \quad \langle 'end-model' \rangle \quad \rightarrow \\
\)

\( \langle \text{Package} \rangle := \langle 'package' \rangle \quad \langle \text{Identifier} \rangle \quad \cdots \\
\quad \langle \text{Directives} \rangle \quad \langle \text{Parameters} \rangle \quad \langle \text{Body} \rangle \quad \langle 'end-package' \rangle \quad \rightarrow \\
\)

\( \langle \text{Directives} \rangle := \langle 'uses' \rangle \quad \langle \text{String} \rangle \\
\quad \langle 'imports' \rangle \quad \langle \text{String} \rangle \\
\quad \langle 'options' \rangle \quad \langle \text{Identifier} \rangle \\
\quad \langle 'version' \rangle \quad \langle \text{Integer} \rangle \quad \langle '.' \rangle \quad \langle \text{Integer} \rangle \quad \langle '.' \rangle \quad \langle \text{Integer} \rangle \\
\)

\( \langle \text{Parameters} \rangle := \langle 'parameters' \rangle \quad \cdots \\
\quad \langle \text{Identifier} \rangle \quad \langle '=' \rangle \quad \langle \text{Expression} \rangle \\
\quad \langle 'end-parameters' \rangle \quad \rightarrow \\
\)

\( \langle \text{Body} \rangle := \langle \text{Declarations} \rangle \\
\quad \langle \text{Requirements} \rangle \\
\quad \langle \text{SubProgram}\_\text{decl} \rangle \\
\quad \langle \text{SubProgram}\_\text{def} \rangle \\
\quad \langle 'include' \rangle \quad \langle \text{String} \rangle \quad \langle \text{Identifier} \rangle \\
\quad \langle \text{Statement} \rangle \\
\)

Mosel Reference Manual
Syntax diagrams for the Mosel language
A.2 Expressions
A.3 Initializations data file format

Syntax diagrams for the Mosel language
Syntax diagrams for the Mosel language

Mosel Reference Manual
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 (driver_error)**
Likely causes are an incorrect access path or write-protected files.

**E-22 I cannot open file ‘file’ for reading (driver_error)**
Likely causes are an incorrect access path or filename or not read-enabled files.

**E-23 Error when writing to the file ‘file’ (driver_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’ (driver_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.

*Examples:*

```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.

*Examples:*

```plaintext
declarations
 D: range
end-declarations

initializations from "test.dat"
 D
end-initializations

Contents of test.dat:

```
[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.

*Examples:*

```plaintext
declarations
 C: set of real
end-declarations

initializations from "test.dat"
 C
end-initializations

Contents of test.dat:

```
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.

*Examples:*

```plaintext
b := 2E -10
```

E must be immediately followed by a signed integer (i.e. no spaces).

E-29 *Wrong file descriptor number for selection (num)*

fselect is used with an incorrect parameter value.

E-34 *I cannot find IO driver ‘driver’*

The system cannot locate the IO driver *driver* for opening a file. This may happen if the driver is provided by a module not already loaded in memory. To avoid this problem the module name should be given with the driver name. For instance use "mmodbc.odbc" instead of "odbc" alone.

E-35 *Error when closing file ‘file’ (driver_error)*

An error occurred while closing a file. Typically the last write operation for clearing buffers failed.
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:*

```mosel
token: )
writeln(3 mod)
```

*mod must be followed by an integer (or a numerical expression evaluating to an integer).*

```mosel
token: write
if i > 0
write("greater")
end-if
then has been omitted.
```

```mosel
token: end
if i > 0 then write("greater") end-if
```

*A semicolon must be added to indicate termination of the statement preceding 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:*

```mosel
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.*

```mosel
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).

*Examples:*

```mosel
procedure myprint(a:integer)
writeln("a: ", a)
end-procedure
```

```mosel
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.  
Examples:

‘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.  
Examples:

```mosel
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’  
Examples:

```mosel
op: and  
2+3 and true
```

E-112 Numeric expression expected for operator ‘op’  
Examples:

```mosel
op: +  
12+(13)
```

```mosel
op: *  
uses "mmxprs"
``` 

```mosel
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. \texttt{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.  
*Examples:*  
\begin{verbatim}
  if("abc" in "acd") then writeln("?"); end-if
\end{verbatim}

The operator \texttt{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 (\textit{i.e.} \texttt{>=} or \texttt{<=} ...) or an equality constraint occurs without decision variables, e.g. \texttt{2=1}

E-122  **Control parameter ‘param’ unknown**  
The control parameters of Mosel are documented in the Mosel Reference manual under function \texttt{getparam}. All control parameters provided by a module, e.g. \texttt{mmxprs}, can be displayed with the command \texttt{EXAM}, e.g. \texttt{exam -p mmxprs}. In IVE this information is displayed by the module browser.

E-123  **‘identifier’ is not defined**  
\texttt{identifier} is used without or before declaring it. Check the spelling of the name. If \texttt{identifier} is defined by a module, make sure that the corresponding module is loaded. If \texttt{identifier} is a subroutine that is defined later in the program, add a forward declaration at the beginning of the model.

E-125  **Set expression expected**  
For instance computing the union between an integer constant and a set of integers: \texttt{union(12+{13})}

E-126  **String expression expected**  
A string is expected here: for instance a file name for an initialization block.

E-127  **A function cannot be of type ‘type’**  
Some types cannot be the return value of a function. Typically no function can return a decision variable (type \texttt{mpvar}).

E-128  **Type ‘type’ has no field named ‘field’**  
Trying to access an unknown field in a record type.  
*Examples:*  
\begin{verbatim}
declarations
  myrec=record
    i,j:integer
  end-record
r:myrec
end-declarations
r.k:=0
\end{verbatim}

\texttt{k} is not a field of \texttt{r}.
E-129  **Type ‘type’ is not a record**
Trying to use a record dereference on an object that is not a record. For instance using `i.j` with `i` defined as an integer.

E-130  **A type definition cannot be local**
It is not possible to declare a type in a procedure or function.

E-132  **Array ‘identifier’ is not indexed by ranges: assignment may be incorrect**
When performing an inline initialization (operator `::`) on an array, it is recommended to list indices if the indexing sets are not ranges. Indeed, since order of set elements is not guaranteed the values provided may not be assigned to the expected cells in the array.

**Examples:**

```
declarations
  a:array({3,2,1}) of integer
end-declarations
! a::[3,2,1] !=> a(1)=3 a(2)=2 a(3)=1
a::([3,2,1])[3,2,1] !=> a(1)=1 a(2)=2 a(3)=3
```

E-132  **Set or list expression expected**
Aggregate operators (like `sum` or `forall`) require sets or lists to describe the domains for their loops.

**Examples:**

```
declarations
  i:integer
end-declarations
forall(i = 2) writeln(i)
```

Since `i` is declared as an integer before the loop, the expression `i=2` is a logical expression (it checks whether `i` is equal to 2) instead of an index definition.

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-149  **Some requirements are not met**
A package may declare `requirements`: these are symbols that must be declared by models using this package. This error occurs when a model uses a package without providing the definitions for all the requirements.

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.

**Examples:**

```
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.

**Examples:**

```
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 section 2.3.3 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.

**Examples:**

```
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.

*Examples:*

```mosel
procedure myproc(F,G,H:array(range) of real, a,b,c:real)
writeln("something")
end-procedure
```

Separate declaration of every array is required:

```mosel
procedure myproc(F:array(range) of real, G:array(range) of real,
H:array(range) of real, a,b,c:real)
```

W-164  A local symbol cannot be made public

*Examples:*

```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.

*Examples:*

```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.

*Examples:*

```mosel
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:*

```mosel
something: :=
a: 3
```

The assignment is indicated by `:=`.

```mosel
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)

Examples:

   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).

   Examples:

   sum(i = 1..10)

   Replace = by in.

E-171 The [upper/lower] bound of a range is not an integer expression

Examples:

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.

E-172 Only a reference to a public set is allowed here

   All index sets of a public array must also be public.

E-173 Statement allowed in packages only

   The block requirements can only be used in packages.

E-175 Index sets of array types must be named

   User types defined as arrays must be indexed by named sets (i.e. declared separately). For instance it is not allowed to use range or set of string as an index of such an array.

E-176 Only a public type is allowed here

   If a user type depending on another user type is declared declared public, the secondary type must also be public. For instance, assuming type T1 is private, it is not possible to declare T2 as a public T2=set of T1.

E-177 Incorrect number of initializers (n1/n2)

   In an inline initialization (operator ::) the number of provided values to assign does not match the list of indices.

E-202 Integer constant expected

   Versions numbers (stated by means of the version compiler directive) must consist in 1 to 3 numbers separated by dots (e.g. 1.2.3). This error is displayed if a version number does not conform to this syntax.
B.2.1 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.2.2 Errors related to packages

E-320 Package ‘package’ not found
A package has not been found in the module path (see section 2.3.1 for the search rules).

E-321 ‘file’ is not a package
Typically displayed if a model is used as a package (the source for the bim file starts with the model keyword instead of package).

E-322 Wrong version for package ‘package’(num1.num2.num3/num4.num5.num6)
A model is compiled with package A depending on a package B. The bim file Mosel has loaded for B is not compatible with the one used for compiling A.

E-323 Package ‘package’ imported several times
A package cannot be imported several times in a model. This error occurs usually when a model uses packages A and B, and package B already includes A.

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 g with the compile, cload, or execute command.

B.3.1 Initializations

E-30 Duplicate label ‘label’ at line num of file ‘file’ (ignored)
The same label is used repeatedly in a data file.

Examples:

D: [1 2 3]
D: [1 2 4]
Error when reading label 'label' at (num1,num2) of file 'file'
The data entry labeled 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.

Error when writing label 'label' at (num1,num2) of file 'file'
The data entry labeled label has not been written correctly. Usually this message is preceded by a more detailed one, e.g. E-23.

Initialization with file 'file' failed for: list_of_identifier
Summary report at the end of an initializations section. Usually this message is preceded by more detailed ones, e.g. E-27, E-28, E-30, E-31.

B.3.2 General runtime errors

Division by zero
Division by 0 resulting from the evaluation of an expression.

Math error performing function ‘identifier’
For example ln used with inadmissible argument, such as 0 or negative values.

Inconsistent range
Typically displayed if the lower bound specified for a range is greater than its upper bound.
Examples:

\[ D := 3 \ldots -1 \]

Conflicting types in set operation (op)
A set operation can only be carried out between sets of the same type.
Examples:

```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.

An index is out of range
An attempt is being made to access an array entry that lies outside of the index sets of the array.

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.

Trying to access an uninitialized object (type_of_object)
Occurs typically in models that define subroutines.
Examples:

```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-1009** *Too many initializers*
The number of data elements exceeds the maximum size of an array.

*Examples:*

```mosel
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.

*Examples:*

```mosel
declarations
x,y: mpvar
end-declarations
c:= x is_integer
```

**E-1013** *Infeasible constraint*  
The simple cases of infeasible unnamed constraints that are detected at run time include:

*Examples:*

```mosel
declarations
x: mpvar
end-declarations
i:=-1
if(i>=0,x,0)>=1
! or:
x-x>=1
```

**E-1014** *Conflicting types in array operation (op)*
An array operation (like assignment) can only be carried out between arrays of the same type and structure.

**E-1015** *Trying to modify a constant list*
Occurs, for instance, when it is attempted to apply a destructive operation (like `splittail`) to a constant list.

**E-1016** *Trying to get an element in an empty set*
The function `getfirst` or `getlast` is applied to an empty set.

**E-1017** *Trying to get an element in an empty list*
The function `getfirst` or `getlast` is applied to an empty list.

**E-1100** *Empty problem*
We are trying to generate or load an empty problem into a solver (i.e. no 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.
B.3.3 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).

B.3.4 Module manager errors

E-350  *Module ‘module’ not found*
A module has not been found in the module path (see section 2.3.1 for the search rules). 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 license 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.num5.num6)*
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).

E-359  *Driver ‘pkg.driver’ rejected (reason)*
A module publishes an IO driver which name is invalid or that is missing some mandatory function.

E-360  *Control parameter ‘module.param’ unknown (setting ignored)*
It is possible to set module parameters when running a model (using the RUN command for instance): in the list of assignments, a control parameter cannot be found in the indicated module.

E-361  *Version number truncated (‘vernum’)*
A version number (for module, model or package) consists in three positive numbers a.b.c. This error is raised if one of these numbers is larger than 999.

Error messages Mosel Reference Manual
<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>comment, sign, skip</td>
<td>4, 10, 28</td>
</tr>
<tr>
<td>commentary</td>
<td>63</td>
</tr>
<tr>
<td>communication</td>
<td>191, 254</td>
</tr>
<tr>
<td>comparator</td>
<td>23</td>
</tr>
<tr>
<td>COMPILE, compile</td>
<td>3, 4</td>
</tr>
<tr>
<td>compiler</td>
<td>130</td>
</tr>
<tr>
<td>compiled, compiler,</td>
<td>12, 12</td>
</tr>
<tr>
<td>compiler options,</td>
<td>13</td>
</tr>
<tr>
<td>concatenated list</td>
<td>23</td>
</tr>
<tr>
<td>condition</td>
<td>22</td>
</tr>
<tr>
<td>connect, connector,</td>
<td>161, 178</td>
</tr>
<tr>
<td>connection number,</td>
<td>158, 175</td>
</tr>
<tr>
<td>constant</td>
<td>19</td>
</tr>
<tr>
<td>compile time, definition, display, run time</td>
<td>19</td>
</tr>
<tr>
<td>constants</td>
<td>38</td>
</tr>
<tr>
<td>constraint, activity</td>
<td>64</td>
</tr>
<tr>
<td>anonymous</td>
<td>27</td>
</tr>
<tr>
<td>coefficient</td>
<td>65</td>
</tr>
<tr>
<td>dual, hide</td>
<td>66, 96</td>
</tr>
<tr>
<td>name</td>
<td>27</td>
</tr>
<tr>
<td>ranging information, right hand side</td>
<td>24</td>
</tr>
<tr>
<td>sensitivity ranges,</td>
<td>276</td>
</tr>
<tr>
<td>set coefficient, set of variables, set type, slack, test hidden, type</td>
<td>95, 82, 99, 78, 84, 24, 80</td>
</tr>
<tr>
<td>context</td>
<td>191, 254</td>
</tr>
<tr>
<td>CONTINUE</td>
<td>7</td>
</tr>
<tr>
<td>control parameter, get, set</td>
<td>14, 73, 97</td>
</tr>
<tr>
<td>conversion basic type</td>
<td>21</td>
</tr>
<tr>
<td>copy</td>
<td>203</td>
</tr>
<tr>
<td>copytext</td>
<td>242</td>
</tr>
<tr>
<td>cos</td>
<td>44</td>
</tr>
<tr>
<td>create directory, create</td>
<td>215, 18, 45</td>
</tr>
<tr>
<td>cross recursion, CT_BIN, CT_CONT, CT_EQ, CT_FREE, CT_GEQ</td>
<td>34, 80, 99, 80, 99, 80, 99, 80, 99</td>
</tr>
<tr>
<td>CT_INT, CT_LEQ, CT_PINT, CT_SEC, CT_SINT, CT_SOS1, CT_SOS2, CT_UNB</td>
<td>80, 99, 80, 99, 80, 99, 80, 99, 80, 99</td>
</tr>
<tr>
<td>currentDate, currentTime</td>
<td>50, 51</td>
</tr>
<tr>
<td>cut add, add array, delete, drop, get active, list from cut pool, load, store, store array</td>
<td>303, 304, 305, 306, 307, 308, 309, 310, 311</td>
</tr>
<tr>
<td>cuthead, cuttail, cuttext</td>
<td>52, 53, 243</td>
</tr>
<tr>
<td>D data input, local, output, read, save</td>
<td>111, 32, 111, 92, 28</td>
</tr>
<tr>
<td>database connect, disconnect, logoff</td>
<td>161, 179</td>
</tr>
<tr>
<td>date, date, datefmt, datetime, datetimefmt</td>
<td>198, 199, 199, 200</td>
</tr>
<tr>
<td>DEBUG, debug mode OCI, ODBC</td>
<td>5, 158, 175</td>
</tr>
<tr>
<td>debugger, declaration</td>
<td>7, 12, 15</td>
</tr>
<tr>
<td>declaration, forward, implicit, list, record, set</td>
<td>34, 26, 17, 18, 16</td>
</tr>
<tr>
<td>declarations, declarative, directives, directory</td>
<td>11, 15, 12, 263, 218</td>
</tr>
<tr>
<td>delete cut, delete directives, model cuts</td>
<td>305, 263, 264</td>
</tr>
</tbody>
</table>

The index entry for cut cuts model, compile, COMPILE, compile, compiled, compiler directives, compiler library, compiler options, concatenation, list, condition, connect, number, connector, constant, compile time, definition, display, run time, constants, activity, anonymous, coefficient, dual, hide, name, ranging information, right hand side, sensitivity ranges, set coefficient, set of variables, set type, slack, test hidden, type, context, continue, control parameter, get, set, conversion, basic type, copy, file, copytext, cos, create, directory, create, create, create, create, cross recursion, CT_BIN, CT_CONT, CT_EQ, CT_FREE, CT_GEQ, CT_INT, CT_LEQ, CT_PINT, CT_SEC, CT_SINT, CT_SOS1, CT_SOS2, CT_UNB, currentDate, currentTime, cut add, add array, delete, drop, get active, list from cut pool, load, store, store array, cuthead, cuttail, cuttext, DEBUG, data input, local, output, read, save, database connect, disconnect, logoff, date, date, datefmt, datetime, datetimefmt, debug mode OCI, ODBC, debugger, declaration, forward, implicit, list, record, set, declarations, declarative, directives, directory, delete cut, delete directives, model cuts.
<table>
<thead>
<tr>
<th>Page</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E-23</td>
<td>320</td>
<td>E-23, 320</td>
</tr>
<tr>
<td>E-24</td>
<td>320</td>
<td>E-24, 320</td>
</tr>
<tr>
<td>E-25</td>
<td>321</td>
<td>E-25, 321</td>
</tr>
<tr>
<td>E-26</td>
<td>321</td>
<td>E-26, 321</td>
</tr>
<tr>
<td>E-27</td>
<td>321</td>
<td>E-27, 321</td>
</tr>
<tr>
<td>E-28</td>
<td>321</td>
<td>E-28, 321</td>
</tr>
<tr>
<td>E-29</td>
<td>321</td>
<td>E-29, 321</td>
</tr>
<tr>
<td>E-30</td>
<td>321</td>
<td>E-30, 321</td>
</tr>
<tr>
<td>E-301</td>
<td>329</td>
<td>E-302, 329</td>
</tr>
<tr>
<td>E-302</td>
<td>329</td>
<td>E-303, 329</td>
</tr>
<tr>
<td>E-303</td>
<td>329</td>
<td>E-307, 329</td>
</tr>
<tr>
<td>E-307</td>
<td>329</td>
<td>E-308, 329</td>
</tr>
<tr>
<td>E-308</td>
<td>329</td>
<td>E-31, 330</td>
</tr>
<tr>
<td>E-31</td>
<td>330</td>
<td>E-32, 330</td>
</tr>
<tr>
<td>E-32</td>
<td>330</td>
<td>E-320, 330</td>
</tr>
<tr>
<td>E-320</td>
<td>330</td>
<td>E-321, 330</td>
</tr>
<tr>
<td>E-321</td>
<td>330</td>
<td>E-322, 330</td>
</tr>
<tr>
<td>E-322</td>
<td>330</td>
<td>E-323, 330</td>
</tr>
<tr>
<td>E-323</td>
<td>330</td>
<td>E-33, 330</td>
</tr>
<tr>
<td>E-33</td>
<td>331</td>
<td>E-34, 331</td>
</tr>
<tr>
<td>E-34</td>
<td>331</td>
<td>E-35, 331</td>
</tr>
<tr>
<td>E-35</td>
<td>331</td>
<td>E-36, 331</td>
</tr>
<tr>
<td>E-36</td>
<td>331</td>
<td>E-37, 331</td>
</tr>
<tr>
<td>E-37</td>
<td>331</td>
<td>E-38, 331</td>
</tr>
<tr>
<td>E-38</td>
<td>331</td>
<td>E-39, 331</td>
</tr>
<tr>
<td>E-39</td>
<td>331</td>
<td>E-40, 331</td>
</tr>
<tr>
<td>E-40</td>
<td>331</td>
<td>E-41, 331</td>
</tr>
<tr>
<td>E-41</td>
<td>331</td>
<td>E-42, 331</td>
</tr>
<tr>
<td>E-42</td>
<td>331</td>
<td>E-43, 331</td>
</tr>
<tr>
<td>E-43</td>
<td>331</td>
<td>E-44, 331</td>
</tr>
<tr>
<td>E-44</td>
<td>331</td>
<td>E-45, 331</td>
</tr>
<tr>
<td>E-45</td>
<td>331</td>
<td>E-46, 331</td>
</tr>
<tr>
<td>E-46</td>
<td>331</td>
<td>E-47, 331</td>
</tr>
<tr>
<td>E-47</td>
<td>331</td>
<td>E-48, 331</td>
</tr>
<tr>
<td>E-48</td>
<td>331</td>
<td>E-49, 331</td>
</tr>
<tr>
<td>E-49</td>
<td>331</td>
<td>E-50, 331</td>
</tr>
<tr>
<td>E-50</td>
<td>331</td>
<td>E-51, 331</td>
</tr>
<tr>
<td>E-51</td>
<td>331</td>
<td>E-52, 331</td>
</tr>
<tr>
<td>E-52</td>
<td>331</td>
<td>E-53, 331</td>
</tr>
<tr>
<td>E-53</td>
<td>331</td>
<td>E-54, 331</td>
</tr>
<tr>
<td>E-54</td>
<td>331</td>
<td>E-55, 331</td>
</tr>
<tr>
<td>E-55</td>
<td>331</td>
<td>E-56, 331</td>
</tr>
<tr>
<td>E-56</td>
<td>331</td>
<td>E-57, 331</td>
</tr>
<tr>
<td>E-57</td>
<td>331</td>
<td>E-58, 331</td>
</tr>
<tr>
<td>E-58</td>
<td>331</td>
<td>E-59, 331</td>
</tr>
<tr>
<td>E-59</td>
<td>331</td>
<td>E-60, 331</td>
</tr>
<tr>
<td>E-60</td>
<td>331</td>
<td>E-61, 331</td>
</tr>
<tr>
<td>E-61</td>
<td>331</td>
<td>E-62, 331</td>
</tr>
<tr>
<td>E-62</td>
<td>331</td>
<td>E-63, 331</td>
</tr>
<tr>
<td>E-63</td>
<td>331</td>
<td>E-64, 331</td>
</tr>
<tr>
<td>E-64</td>
<td>331</td>
<td>E-65, 331</td>
</tr>
<tr>
<td>E-65</td>
<td>331</td>
<td>E-66, 331</td>
</tr>
<tr>
<td>E-66</td>
<td>331</td>
<td>E-67, 331</td>
</tr>
<tr>
<td>E-67</td>
<td>331</td>
<td>E-68, 331</td>
</tr>
<tr>
<td>E-68</td>
<td>331</td>
<td>E-69, 331</td>
</tr>
<tr>
<td>E-69</td>
<td>331</td>
<td>E-70, 331</td>
</tr>
<tr>
<td>E-70</td>
<td>331</td>
<td>E-71, 331</td>
</tr>
<tr>
<td>E-71</td>
<td>331</td>
<td>E-72, 331</td>
</tr>
<tr>
<td>E-72</td>
<td>331</td>
<td>E-73, 331</td>
</tr>
<tr>
<td>E-73</td>
<td>331</td>
<td>E-74, 331</td>
</tr>
<tr>
<td>E-74</td>
<td>331</td>
<td>E-75, 331</td>
</tr>
<tr>
<td>E-75</td>
<td>331</td>
<td>E-76, 331</td>
</tr>
<tr>
<td>E-76</td>
<td>331</td>
<td>E-77, 331</td>
</tr>
<tr>
<td>E-77</td>
<td>331</td>
<td>E-78, 331</td>
</tr>
<tr>
<td>E-78</td>
<td>331</td>
<td>E-79, 331</td>
</tr>
<tr>
<td>E-79</td>
<td>331</td>
<td>E-80, 331</td>
</tr>
<tr>
<td>E-80</td>
<td>331</td>
<td>E-81, 331</td>
</tr>
<tr>
<td>E-81</td>
<td>331</td>
<td>E-82, 331</td>
</tr>
<tr>
<td>E-82</td>
<td>331</td>
<td>E-83, 331</td>
</tr>
<tr>
<td>E-83</td>
<td>331</td>
<td>E-84, 331</td>
</tr>
<tr>
<td>E-84</td>
<td>331</td>
<td>E-85, 331</td>
</tr>
</tbody>
</table>

- **elementary type**: 16
- **elif**: 11, 29
- **else**: 11, 29, 30
- **end**: 11
- **end-case**: 30
- **end-declarations**: 15
- **end-function**: 33
- **end-if**: 29
- **end-model**: 12
- **end-package**: 12
- **end-procedure**: 33

**enumerate**
- quadratic terms**: 197

**environment variable**: 207, 208

**eof**: 83

**EP_MAX**: 49, 189
- **EP_MIN**: 49, 189
- **EP_MPS**: 49, 189
- **EP_STRIP**: 49, 189

**error**
- detection**: 12
- **ODBC**: 174

**error control**
- **IO**: 61, 73, 97

**escape sequence**: 22

**escape sequences**: 22

**ETC_APPEND**: 111
- **ETC_DENSE**: 111
- **ETC_IN**: 111
- **ETC_NOQ**: 111
- **ETC_NOZEROS**: 111
- **ETC_OUT**: 111
- **ETC_SGLQ**: 111
- **ETC_SPARSE**: 111
- **ETC_TRANS**: 111

**evaluation**: 11

**event**
- class**: 149
- drop next**: 145
- get next**: 144
- null**: 147
- queue**: 146
- send**: 141
- sender ID**: 148
- value**: 150
- wait for**: 142

**event class**: 140
- wait for**: 143

**event queue**: 128

**event value**: 140

**EXAMINE**: 6

**excel**: 185

**EXEC**: 5

**exists**: 46

**exit**: 47

**exit code**
- **model**: 136

**exp**: 48

**explterm**: 11, 13

**exponential function**: 48

**export**
- **problem**: 49
- **quadratic problem**: 189

**EXPORTPROB**: 5

**exportprob**: 49, 189

**expression**: 20
- linear constraint**: 24
- list**: 23
- print**: 107
- set**: 23
- set type**: 99
- string**: 22
- **string terminator**: 11

**type**: 21

**extended syntax**: 176

**F**
- **F_APPEND**: 61
- **F_ERROR**: 68
- **F_INPUT**: 55, 61, 67, 68
- **F_LINBUF**: 61
- **F_OUTPUT**: 55, 61, 67, 68
- **F_SILENT**: 61

**failure**: 159, 177

**false**: 11, 16

**fclose**: 37, 55

**fcopy**: 203

**fdelete**: 204
fflush, 37, 56
file
  access mode, 209
append, 61
close, 55
copy, 203
delete, 204
ID, 67
in/output, 111
inclusion, 14
initialization, 28
IO, 37
move, 205
name, 68
open, 61
read, 36, 92
rename, 205
select, 62
size, 210
status, 209
time, 211
write, 107
file extension, 3
files, 12
finalize
  set, 57
finalize, 18, 57
findfirst, 58
findlast, 59
findtext, 247
fix
  variable, 268
fixglobal, 268
floor, 60
flush buffer, 56
FLUSHLIBS, 6
fmove, 205
fopen, 37, 61
forall, 11, 30
format string, 104
forward, 11, 34
free
  info table, 196
from, 11
fselect, 37, 62
fskipline, 37, 61, 63
function
  return value, 32
type, 32
function, 11
function call, 20
G
get
  active cuts, 307
cuts from cut pool, 308
getact, 64
goingnumber, 229
getbstat, 269
getchar, 248
getclass, 149
getcnlist, 307
getcoeff, 65
getcplist, 308
getcwd, 206
getdate, 214
getday, 221
getdual, 66
getenv, 207
getexitcode, 136
getfid, 37, 67
getfirst, 69
getfname, 68
getfromid, 148
getsize, 210
getfstat, 209
gftime, 211
gethead, 70
gethour, 225
getid, 134
getiiis, 270
getinfeas, 271
getlast, 71
getlb, 272
getminute, 226
getmonth, 222
getmsec, 228
getname, 273
getnextevent, 144
getobjval, 72
getparam, 61, 73, 157, 174, 199, 259, 274
getprobsat, 274
getqexpnextterm, 197
getqexpso, 194
getqexpstat, 195
getrange, 275
getrcost, 74
getreadcnt, 75
getreverse, 76
getsecond, 227
getsensrng, 276
getsize, 77, 241
getsllack, 78
getsol, 79, 190, 194
getstatus, 135
getsysstat, 212
gettail, 81
gftime, 213
gettxtbuf, 257
gettxtsize, 256
gettype, 80
getub, 277
getvalue, 150
getvars, 82
getweekday, 224
getyear, 223
global search
  reset, 278
graph, 113
graphical interface, 2
H
help, 2
hidden constraint, 84, 96
I
ID
event sender, 148
file, 67
model, 134
stream, 67
identifier, 10
if, 11, 21, 29
IIS, see irreducible infeasible set
IMCI, 191, 254
imports, 11, 12
in, 11, 24
include, 11, 14
indexing set, 17
INFO, 3
info
    quadratic expression, 195
initglobal, 278
initialisations, 11
initializations, 11
inline initialization, 26
input file, 111
input stream, 55, 61, 62, 126, 132
    read, 92
test eof, 83
InputStream, 126
inserttext, 246
integer
    read, 164, 181
integer, 11, 16, 21
integrality check, 279
inter, 11, 23
inter-module communication, 191, 254
interactive mode, 3
interface
    inter-module communication, 191, 254
interpreted, 12
intersection, 23
IO
    error, 61, 73, 97
status, 61, 73
switching between streams, 37
IO driver, 169
diskdata, 112
Excel, 185
Java, 126
ODBC, 185
shmem, 151
ioctl, 61, 73, 97
iostatus, 61, 73
irreducible infeasible set, 270
is_binary, 11, 24
is_continuous, 11, 24
is_free, 11, 24
is_integer, 11, 24
is_partint, 11, 24
is_semcont, 11, 24
is_semint, 11, 24
is_sos1, 11, 24
is_sos2, 11, 24
iseof, 83
ishidden, 84
isintegral, 279
isodd, 85
isqueueempty, 146
isvalid, 230
item
    number read, 73
iterator, 22
IVE_BLACK, 115
IVE_BLUE, 115
IVE_CYAN, 115
IVE_GREEN, 115
IVE_MAGENTA, 115
IVE_RED, 115
IVE_RGB, 114
IVE_WHITE, 115
IVE_YELLOW, 115
IVEaddplot, 115
IVEdrawarrow, 116
IVEdrawellipse, 125
IVEdrawgant, 117
IVEdrawlabel, 118
IVEdrawline, 119
IVEdrawpoint, 120
IVEdrawrectangle, 124
IVEerase, 121
IVEpause, 122
IVEzoom, 123

J
Java
    IO drivers, 126
java, 126
jraw, 126

K
keepassert, 13, 41
keyword
    SQL, 174
keywords, 6, 11

L
language, 10
largest value, 89
length
    string, 157, 175
library
    Run Time, 1
lincr, 11, 16, 187, 188, 191
line breaking, 11
line control directive, 15
lines
    number affected, 159, 176
    number transferred, 159, 177
LIST, 4, 7
list, 17
    compare, 24
    constant, 23
    finalize, 57
    find element, 58, 59
    first element, 69
    head, 70
    last element, 71
    remove elements, 52, 53
    reverse, 76, 93
    size, 77
    split, 101, 102
tail, 81
list, 11
ln, 86
LOAD, 4
load
  basis, 280
cut, 309
model, 4
module, 12
package, 12, 13
problem, 282
load, 131
loadbasis, 280
loadcuts, 309
loadmipsol, 281
loadprob, 187, 282
log, 87
logarithm
  base 10, 87
  natural, 86
logical and, 24
logical negation, 24
logical or, 24
loop, 30
loop statement, 25
lower bound, 272
  set, 294
LP format, 5
  maximization, 49
  minimization, 49
LSDRVS, 6
LSLIBS, 6
M
M_E, 38
M_PI, 38
makedir, 215
makesos1, 24, 88
makesos2, 24, 88
matrix
  column order, 260
  matrix output, 5
max, 11, 22
MAX_INT, 38
MAX_REAL, 38
maximize, 187, 284
maximum value, 22, 89
maxlist, 89
message printing, 159, 177
  Optimizer, 260
  min, 11, 22
  minimize, 187, 284
  minimum value, 22, 90
minlist, 90
MIP solution
  load, 281
  save, 289
mod, 11, 22
Model, 129
model
  active, 4
  body, 12
  compile, 130
  debug, 5
  execute, 4
  exit code, 136
ID, 134
load, 131
name, 4
  pause execution, 122
  profile, 5
  reset, 5, 138
  run, 133
  sequence number, 4
  size, 4
  source, 1
  status, 135
  stop, 137
  structure, 12
  unload, 139
  version, 4
model, 4, 11, 12
model cut, 296
delete, 264
model management, 129
model manager, 1
model parameter, 14
model_version, 73
module, 2
  dependency, 191, 255
module structure
  advantages, 2
  modules, 10
  monthnames, 200
mosel, 2
debugger, 7
  main commands, 2
Mosel compiler, 1
Mosel Console, 2
move
  file, 205
  MP type, 16
  MPS format, 5, 49
mpvar, 11, 16, 187, 191
N
name
  scramble, 49
  variable, 273
names
  loading, 260
nbread, 73, 92
new line, 22
NEXT, 7
next, 11, 31
noimplicit, 13, 26
non-relational, 158, 176
not, 11, 24
not in, 24
nullevent, 147
number
  connection, 158, 175
  lines, 159, 176, 177
O
objective value, 72
OCI
debug mode, 158
  IO driver, 169
oci, 169
OCIautocommit, 157
OCIbufsize, 157
OCIcolsize, 157
OCIcommit, 167
OCIconnection, 158, 161
OCIdebug, 158
OCIexecute, 167
OCIlogoff, 162
OCIlogon, 161
OCIrowcnt, 159
OCIrowxfr, 159
OCIsuccess, 159
OCIverbose, 159
ODBC
db mode, 175
IO driver, 185
cdbc, 185
odd number, 85
of, 11
open
file, 61
stream, 61
openpipe, 216
operation
elementary, 25
operator, 20
arithmetic, 22
optimization
direction, 5
Optimizer
loading names, 260
message printing, 260
problem name, 260
problem pointer, 260
optimizer problem status, 274
OPTION, 6
options, 11, 12
or, 11, 24
output file, 111
output stream, 55, 61, 62, 126, 132
flush, 56
write, 107
OutputStream, 126
P
package, 35
structure, 12
package, 11, 12
parameter, 14, 73, 97
parameters, 12
parameters, 11
parent model, 141
parseextn, 253
parseint, 251
parser parameter, 73
parser_date, 73
parser_file, 73
parser_line, 73
parser_time, 73
parser.UTCdate, 73
parser.UTCtime, 73
parser_version, 73
parsereal, 252
pastext, 245
pause
model execution, 122
plot
add, 115
primal solution, 79
PRINT, 5
print, 107
problem, 49
quadratic problem, 189
printing, 159, 177
printing format, 73, 97
private symbol, 35
problem, 5
export, 49
load, 282
maximize, 284
print, 49
status, 274
unload, 283
write, 301
problem name
Optimizer, 260
problem pointer
Optimizer, 260
procedural, 12
procedure, 27
body, 32
procedure, 11
procedures
passing of formal parameters, 33
prod, 11, 22
product, 22
PROFILE, 5
public, 11, 35
Q
qexp, 187, 188
qsort, 217
quadratic expression
enumerate terms, 197
get info, 195
solution, 190
quadratic problem
export, 189
print, 189
QUIT, 3, 7
quote, 22
R
random, 91
random number, 91, 98
range, 17
first element, 69
last element, 71
range, 11
range set, 23
ranging information, 275
read
basis, 285
directives, 286
integer value, 164, 181
number of items, 73
real value, 165, 182
string, 166, 183
write, 299
read, 37, 61, 73, 92
readbasis, 285
readcnt, 73, 75, 97
readdirs, 286
readln, 37, 61, 92
readdirs, 286
readln, 37, 61, 92
real
printing format, 73, 97
read, 165, 182
read, 11, 16, 21
realfmt, 73, 97
record, 18
dereference, 20
record, 11
recursion, 32
reduced cost value, 74
remove
directory, 218
removedir, 218
rename
file, 205
repeat, 11, 31
requirement, 36
requirements, 11
RESET, 5
reset
global search, 278
reset, 138
resetbasis, 287
returned, 32
reverse, 93
round, 94
rounding, 43, 60, 94
RUN, 4
run, 133
running time, 213
S
save
basis, 288
Optimizer status, 290
savebasis, 288
savemipsol, 289
savestate, 290
secure vectors, 266
SELECT, 4
select
file, 62
stream, 62
selection statement, 25, 29
send, 141
sensivity ranges, 276
service
inter-module communication, 191, 254
module dependency, 191, 255
set, 16
callback, 292
compare, 23
constant, 18
finalize, 57
fixed, 18
in/output, 111
size, 77
set, 11
setbstat, 291
setcallback, 292
setchar, 249
setcoeff, 95
setdate, 239
setday, 231
setdefstream, 132
setenv, 208
sethidden, 96
sethour, 234
setlb, 294
setminute, 235
setmipdir, 295
setmodcut, 296
setmonth, 232
setmsec, 237
setparam, 97, 157, 174, 199, 259
setrandseed, 98
setsecond, 236
SETSTREAM, 3
settime, 238
settype, 99
setub, 297
setucbdata, 298
setyear, 233
shmem, 151
silent, 2
sin, 100
size
array, 77
file, 210
list, 77
set, 77
skip
comment, 63
slack value, 78
sleep, 219
smallest value, 90
solution value, 79, 190
sorting, 217
SOS, 88
declaration, 24
set type, 99
type, 80
special ordered set, 88
splithead, 101
splittail, 102
SQL command
execute, 163, 180
update, 184
SQLbufsize, 175
SQLcolsize, 175
SQLconnect, 178
SQLconnection, 175, 178
SQLdebug, 175
SQLdisconnect, 179
SQLdm, 176
SQLexecute, 180
<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLextn</td>
<td>176</td>
</tr>
<tr>
<td>SQLndxcol</td>
<td>176</td>
</tr>
<tr>
<td>SQLreadinteger</td>
<td>181</td>
</tr>
<tr>
<td>SQLreadreal</td>
<td>182</td>
</tr>
<tr>
<td>SQLreadstring</td>
<td>183</td>
</tr>
<tr>
<td>SQLrowcnt</td>
<td>176</td>
</tr>
<tr>
<td>SQLrowxfr</td>
<td>177</td>
</tr>
<tr>
<td>SQLsuccess</td>
<td>177</td>
</tr>
<tr>
<td>SQLupdate</td>
<td>184</td>
</tr>
<tr>
<td>SQLverbose</td>
<td>177</td>
</tr>
<tr>
<td>sqrt</td>
<td>103</td>
</tr>
<tr>
<td>square root</td>
<td>103</td>
</tr>
<tr>
<td>statement</td>
<td>25</td>
</tr>
<tr>
<td>status</td>
<td>159</td>
</tr>
<tr>
<td>directory</td>
<td>209</td>
</tr>
<tr>
<td>file</td>
<td>209</td>
</tr>
<tr>
<td>IO</td>
<td>61</td>
</tr>
<tr>
<td>model</td>
<td>135</td>
</tr>
<tr>
<td>problem</td>
<td>274</td>
</tr>
<tr>
<td>save</td>
<td>290</td>
</tr>
<tr>
<td>system</td>
<td>212</td>
</tr>
<tr>
<td>STEP</td>
<td>7</td>
</tr>
<tr>
<td>stop</td>
<td>137</td>
</tr>
<tr>
<td>stopoptimize</td>
<td>267</td>
</tr>
<tr>
<td>store</td>
<td></td>
</tr>
<tr>
<td>array of cuts</td>
<td>311</td>
</tr>
<tr>
<td>cut</td>
<td>310</td>
</tr>
<tr>
<td>storecut</td>
<td>310</td>
</tr>
<tr>
<td>storecuts</td>
<td>311</td>
</tr>
<tr>
<td>stream</td>
<td></td>
</tr>
<tr>
<td>close</td>
<td>55</td>
</tr>
<tr>
<td>ID</td>
<td>67</td>
</tr>
<tr>
<td>input</td>
<td>132</td>
</tr>
<tr>
<td>open</td>
<td>61</td>
</tr>
<tr>
<td>output</td>
<td>132</td>
</tr>
<tr>
<td>select</td>
<td>62</td>
</tr>
<tr>
<td>strfmt</td>
<td>104</td>
</tr>
<tr>
<td>string</td>
<td></td>
</tr>
<tr>
<td>formatted</td>
<td>104</td>
</tr>
<tr>
<td>get substring</td>
<td>105</td>
</tr>
<tr>
<td>maximum length</td>
<td>157</td>
</tr>
<tr>
<td>175</td>
<td></td>
</tr>
<tr>
<td>read</td>
<td>166</td>
</tr>
<tr>
<td>183</td>
<td></td>
</tr>
<tr>
<td>string, 11, 16, 21</td>
<td></td>
</tr>
<tr>
<td>string expression</td>
<td></td>
</tr>
<tr>
<td>compare</td>
<td>23</td>
</tr>
<tr>
<td>strip</td>
<td>35</td>
</tr>
<tr>
<td>subroutine</td>
<td>32</td>
</tr>
<tr>
<td>subset</td>
<td>23</td>
</tr>
<tr>
<td>substr</td>
<td>105</td>
</tr>
<tr>
<td>success</td>
<td>159</td>
</tr>
<tr>
<td>suffix notation</td>
<td>5</td>
</tr>
<tr>
<td>34</td>
<td></td>
</tr>
<tr>
<td>sum</td>
<td>11</td>
</tr>
<tr>
<td>22</td>
<td></td>
</tr>
<tr>
<td>summation</td>
<td>22</td>
</tr>
<tr>
<td>superset</td>
<td>23</td>
</tr>
<tr>
<td>symbol</td>
<td></td>
</tr>
<tr>
<td>declaration</td>
<td>32</td>
</tr>
<tr>
<td>import</td>
<td>12</td>
</tr>
<tr>
<td>symbol table</td>
<td>35</td>
</tr>
<tr>
<td>SYMBOLS</td>
<td>6</td>
</tr>
<tr>
<td>synchronization</td>
<td>128</td>
</tr>
<tr>
<td>mechanism</td>
<td></td>
</tr>
<tr>
<td>syntax</td>
<td>10</td>
</tr>
<tr>
<td>SYS_DIR</td>
<td>209</td>
</tr>
<tr>
<td>SYS_DOWN</td>
<td>217</td>
</tr>
<tr>
<td>sys_endparse</td>
<td>201</td>
</tr>
<tr>
<td>SYS_EXEC</td>
<td>209</td>
</tr>
<tr>
<td>sys_fillchar</td>
<td>201</td>
</tr>
<tr>
<td>SYS_MOD</td>
<td>209</td>
</tr>
<tr>
<td>SYS_OTH</td>
<td>209</td>
</tr>
<tr>
<td>SYS_READ</td>
<td>209</td>
</tr>
<tr>
<td>SYS_REG</td>
<td>209</td>
</tr>
<tr>
<td>SYS_TYP</td>
<td>209</td>
</tr>
<tr>
<td>SYS_UP</td>
<td>217</td>
</tr>
<tr>
<td>SYS_WRITE</td>
<td>209</td>
</tr>
<tr>
<td>SYSTEM</td>
<td>3</td>
</tr>
<tr>
<td>system</td>
<td>220</td>
</tr>
<tr>
<td>system command</td>
<td>220</td>
</tr>
<tr>
<td>system status</td>
<td>212</td>
</tr>
<tr>
<td>T</td>
<td></td>
</tr>
<tr>
<td>table of symbols</td>
<td>35</td>
</tr>
<tr>
<td>term</td>
<td></td>
</tr>
<tr>
<td>enumerate</td>
<td>197</td>
</tr>
<tr>
<td>termination</td>
<td>3</td>
</tr>
<tr>
<td>47</td>
<td></td>
</tr>
<tr>
<td>test</td>
<td></td>
</tr>
<tr>
<td>bits</td>
<td>42</td>
</tr>
<tr>
<td>eof</td>
<td>83</td>
</tr>
<tr>
<td>hidden constraint</td>
<td>84</td>
</tr>
<tr>
<td>text</td>
<td>198</td>
</tr>
<tr>
<td>textfmt</td>
<td>240</td>
</tr>
<tr>
<td>then</td>
<td>11</td>
</tr>
<tr>
<td>time</td>
<td>51</td>
</tr>
<tr>
<td>file</td>
<td>211</td>
</tr>
<tr>
<td>time, 198</td>
<td></td>
</tr>
<tr>
<td>time measure</td>
<td>213</td>
</tr>
<tr>
<td>timefmt</td>
<td>200</td>
</tr>
<tr>
<td>timestamp</td>
<td>106</td>
</tr>
<tr>
<td>106</td>
<td></td>
</tr>
<tr>
<td>to</td>
<td>11</td>
</tr>
<tr>
<td>tolerance</td>
<td></td>
</tr>
<tr>
<td>zero</td>
<td>73</td>
</tr>
<tr>
<td>97</td>
<td></td>
</tr>
<tr>
<td>transaction</td>
<td></td>
</tr>
<tr>
<td>commit</td>
<td>167</td>
</tr>
<tr>
<td>rollback</td>
<td>168</td>
</tr>
<tr>
<td>trigonometric functions</td>
<td>40</td>
</tr>
<tr>
<td>44, 100</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>11</td>
</tr>
<tr>
<td>16, 21</td>
<td></td>
</tr>
<tr>
<td>txtresize</td>
<td>258</td>
</tr>
<tr>
<td>type</td>
<td></td>
</tr>
<tr>
<td>constraint</td>
<td>80</td>
</tr>
<tr>
<td>99</td>
<td></td>
</tr>
<tr>
<td>SOS</td>
<td>80</td>
</tr>
<tr>
<td>99</td>
<td></td>
</tr>
<tr>
<td>variable</td>
<td>24</td>
</tr>
<tr>
<td>80, 99</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
</tr>
<tr>
<td>unconstrained</td>
<td>24</td>
</tr>
<tr>
<td>UNDISPLAY</td>
<td>7</td>
</tr>
<tr>
<td>union</td>
<td>23</td>
</tr>
<tr>
<td>union, 11, 23</td>
<td></td>
</tr>
<tr>
<td>UNLOAD</td>
<td>4</td>
</tr>
<tr>
<td>unload</td>
<td></td>
</tr>
<tr>
<td>model</td>
<td>4</td>
</tr>
<tr>
<td>unload</td>
<td></td>
</tr>
<tr>
<td>prob</td>
<td>283</td>
</tr>
<tr>
<td>until</td>
<td>11</td>
</tr>
<tr>
<td>31</td>
<td></td>
</tr>
<tr>
<td>UP</td>
<td>8</td>
</tr>
<tr>
<td>upper bound</td>
<td>277</td>
</tr>
<tr>
<td>set</td>
<td>297</td>
</tr>
<tr>
<td>user comment</td>
<td>4</td>
</tr>
<tr>
<td>user graph</td>
<td>113</td>
</tr>
</tbody>
</table>
add plot, 115
color, 114
draw arrow, 116
draw ellipse, 125
draw Gantt chart, 117
draw label, 118
draw line, 119
draw point, 120
draw rectangle, 124
erase, 121
scale, 123
user type, 19
definition, 19
uses, 11, 12
UTC, 73, 97
V
value
event, 140
variable, 16
check integrality, 279
environment, 207, 208
fix, 268
lower bound, 272
name, 273
ranging information, 275
reduced cost, 74
sensitivity ranges, 276
set coefficient, 95
set lower bound, 294
set type, 99
set upper bound, 297
solution, 79
type, 80
upper bound, 277
version, 4, 11, 12
version number, 2
visual environment, 2
W
W-121, 324
W-152, 325
W-164, 327
W-165, 327
W-166, 327
W-304, 329
W-306, 329
W-85, 332
wait, 142
waitfor, 143
WHERE, 8
while, 11, 31
working directory, 206
write
directives, 300
problem, 301
write, 37, 61, 107, 191
writebasis, 299
writedirs, 300
writeln, 37, 61, 107, 191
writeprob, 301
X
Xpress-IVE, 2, 113
Xpress-Optimizer, 2
XPRSBAR, 284
XPRSCBBARLOG, 292
XPRSCBCHGBRANCH, 292
XPRSCBCHGNODE, 292
XPRSCBCUTLOG, 292
XPRSCBCUTMGR, 292
XPRSCBFREECUTMGR, 292
XPRSCB_GLOBALLOG, 292
XPRSCBINFNODE, 292
XPRSCBINITCUTMGR, 292
XPRSCBINTSOL, 292
XPRSCB_LPLOG, 292
XPRSCBNODECUTOFF, 292
XPRSCBOPTNODE, 292
XPRSCBPRENODE, 292
XPRSCCOLORDER, 260
XPRSDN, 276, 295
XPRSDUAL, 284
XPRSGLB, 284
XPRSIMF, 274
XPRSLCOST, 275
XPRSLIN, 284
XPRSLOACT, 275
XPRSLoadnames, 260
XPRSNET, 284
XPRSNIG, 284
XPRSOPT, 274
XPRSOTH, 274
XPRSPD, 295
XPRSPR, 295
XPRSPRI, 284
XPRSPROBLEM, 260
XPRSPROBNAME, 260
XPRSPU, 295
XPRSTOPCTRLC, 267
XPRSTOPINTERLIMIT, 267
XPRSTOPMIPGAP, 267
XPRSTOPNODELIMIT, 267
XPRSTOPSOLLIMIT, 267
XPRSTOP_TIMELIMIT, 267
XPRSTOP_USER, 267
XPRSTOP, 284
XPRSUCCOST, 275
XPRSUDDN, 275
XPRSUONE, 274
XPRSUUNF, 274
XPRSUUP, 276, 295
XPRSUFACT, 275
XPRSUUP, 275
XPRSVETOPE, 260
Z
zero tolerance, 73, 97
zerotol, 73, 97