Quick reference

FICO® Xpress Optimization

Mosel Language

Quick reference

Last update 29 August, 2017

FICO® Xpress Mosel
Last Revised: 29 August, 2017

How to Contact the Xpress Team

Information, Sales and Licensing

USA, CANADA AND ALL AMERICAS

Email: XpressSalesUS@fico.com

WORLDWIDE

Email: XpressSalesUK@fico.com

Tel: +44 207 940 8718
Fax: +44 870 420 3601

Xpress Optimization, FICO
FICO House
International Square
Starley Way
Birmingham B37 7GN
UK

Product Support

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

Telephone:

NORTH AMERICA
Tel (toll free): +1 (877) 4FI-SUPP
Fax: +1 (402) 496-2224

EUROPE, MIDDLE EAST, AFRICA
Tel: +44 (0) 870-420-3777
UK (toll free): 0800-0152-153
South Africa (toll free): 0800-996-153
Fax: +44 (0) 870-420-3778

ASIA-PACIFIC, LATIN AMERICA, CARIBBEAN
Tel: +1 (415) 446-6185
Brazil (toll free): 0800-891-6146

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

FICO® Xpress Optimization

Mosel Language

Quick reference

29 August, 2017

Contents

1 Mathematical Programming basics

```
model "Chess 1"
 uses "mmxprs"                    ! Use Xpress Optimizer for solving

 declarations
  xs, xl: mpvar                   ! Decision variables
 end-declarations

 Time:= 3*xs + 2*xl <= 160        ! Constraint: limit on working hours
 Wood:= xs + 3*xl <= 200          ! Constraint: raw mat. availability

 xs is_integer; xl is_integer     ! Integrality constraints

 maximize(5*xs + 20*xl)           ! Objective: maximize total profit

 writeln("Solution: ", getobjval) ! Print objective function value

 writeln("small: ", getsol(xs))   ! Print solution for xs
 writeln("large: ", getsol(xl))   ! and xl

 write("Time: ", getact(Time))    ! Constraint activity
 writeln(" ", getslack(Time))     ! and slack

end-model
```

1.1 Decision variables

```
declarations
  x, b, d: mpvar
  ifmake: array(1..10, 1..20) of mpvar
  y, z: array(1..10) of mpvar
end-declarations
```

`mpvar` means *mathematical programming variable* or *decision variable*, sometimes also just called *variable*. Decision variables are unknowns: they have no value until the model is run, and the optimizer finds values for the decision variables.
Variables can take values between 0 and infinity by default, other bounds may be specified:

```
x <= 10
y(1) = 25.5
y(2) is_free
z(2,3) >= -50
z(2,3) <= 50
```

Integer programming types are defined as unary constraints on previously declared decision variables

```
b is_binary                     ! Single binary variable
forall(p in PRODS,l in LINES)
  ifmake(p,l) is_binary         ! An array of binaries
d is_integer                    ! An integer variable
d <= 25                         ! Upper bound on the variable
x is_partint 10                 ! Partial integer (integers up to 10, continuous beyond)
y(3) is_semcont 5               ! Semi-continuous (0 or greater or equal 5)
```

1.2 Constraints

Constraint are declared just like decision variables, in LP/MIP problems they have type `linctr` – linear constraint.

```
declarations
  MaxCap: linctr
  Inven: array(1..10) of linctr
end-declarations
```

The "value" of a constraint entity is a linear expression of decision variables, a constraint type ($\leq$, $\geq$, =), and a constant term. It is set using an assignment statement:

```
MaxCap := 10*x + 20*y + 30*z <= 100
Ctr(3) := 4*x(1) - 3*x(2) >= 10
Inven(2) := stock(2) = stock(1) + buy(2) - sell(2)
```

1.3 Objective function

An objective function is just a constraint with no constraint type.

```
declarations
  MinCost: linctr
end-declarations

MinCost := 10*x(1) + 20*x(2) + 30*x(3) + 40*x(4)
```

1.4 Optimization

```
minimize(MinCost)

maximize(MaxProfit)
```

1.5 Viewing the matrix

After defining the problem the matrix can be output to a file, to examine off line.

Specify LP format for constraint oriented file:

```
exportprob(EP_MIN, "explout", MinCost)
```

Useful Optimizer control settings:

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

1.6 Viewing the solution

Always check the solution status of the problem before accessing any solution values.

```
if (getprobstat=XPRS_OPT) then
  writeln('optimal!')
else
  writeln('not optimal!')
  exit(1)
end-if
```

Alternatively, testing all problem states:

```
case getprobstat of
```

```
  XPRS_OPT: writeln('optimal')
  XPRS_INF: writeln('infeasible')
  XPRS_UNB: writeln('unbounded')
  XPRS_UNF: writeln('unfinished')
else
  writeln('unexpected problem status!')
end-case
```

Accessing the solution values within the model:

```
writeln('Maximum revenue: $', getobjval)
writeln('x(1) = ', getsol(x(1)), '  x(2) = ', x(2).sol)
```

Solution values of constraints: activity value + slack value = RHS

	`MaxCap := 10*x + 20*y <= 30`
Activity value:	`getsol(10*x + 20*y)` `getact(MaxCap)`
Slack value:	`getsol(30 - (10*x + 20*y))` `getslack(MaxCap)`

Xpress Workbench: assuming that the model runs successfully, the logging pane at the bottom of the workspace reports that the run is complete. If a model has been run through the debugger, you can browse solution values of decision variables and constraints in the *Debugger* tab on the right side of the workspace.

2 Data handling basics

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

 declarations
  R = 1..2                              ! Index range
  DUR, WOOD, PROFIT: array(R) of real   ! Coefficients
  x: array(R) of mpvar                  ! Array of variables
 end-declarations

 DUR   :: [3, 2]                        ! Initialize data arrays
 WOOD  :: [1, 3]
 PROFIT:: [5, 20]

 sum(i in R) DUR(i)*x(i) <= 160         ! Constraint definition
 sum(i in R) WOOD(i)*x(i) <= 200
 forall(i in R) x(i) is_integer

 maximize(sum(i in R) PROFIT(i)*x(i))
 writeln("Solution: ", getobjval)
end-model
```

2.1 Data types

Constant data

```
declarations
  NWEEKS = 20
  NDAYS = 7*NWEEKS
  CONV_RATE = 1.425
  DATA_DIR = 'c:\data'
end-declarations
```

Variable data

Declaration

```
declarations
  NPROD: integer
  SCOST: real
  MAXREFVEG: real
  DIR: string
  IF_DEBUG: boolean
  HARD: array(1..5) of real
  COST: array(1..3,1..4) of real
end-declarations
```

Initialization

```
NPROD := 20
SCOST := 5
MAXREFVEG := 200.0
DIR := 'c:\data'
IF_DEBUG := true
HARD :: [8.8, 6.1, 2.0, 4.2, 5.0]
COST :: [11, 12, 13, 14,
         21, 22, 23, 24,
         31, 32, 33, 34]
```

2.2 Sums and loops

Summations

Sum up an array of variables in a constraint:

```
MaxCap := sum(p in 1..10) buy(p) <= 100

MaxCap := sum(p in 1..10) (buy(p) + sum(r in 1..5) make(p,r)) <= 100

MaxCap := sum(p in 1..NP, t in 1..NT)
          CAP(p)*buy(p,t) <= MAXCAP

MaxCap := sum(p in 1..NP) (2*CAP(p)*buy(p)/10 +
          SCAP(p)*sell(p)) <= MAXCAP
```

Loops

Use a loop to assign an array of constraints:

```
forall(t in 2..NT)
  Inven(t) := bal(t) = bal(t-1) + buy(t) - sell(t)
```

Use do/end-do to group several statements into one loop

```
forall(t in 1..NT) do
  MaxRef(t) := sum(i in 1..NI) use(i,t) <= MAXREF(t)

  Inven(t) := store(t) = store(t-1) + buy(t) - use(t)
end-do
```

Can nest forall statements:

```
forall(t in 1..NT) do
  MaxRef(t) := sum(i in 1..NI) use(i,t) <= MAXREF(t)

  forall(i in 1..NI)
    Inven(i,t) := store(i,t) = store(i,t-1) + buy(i,t) - use(i,t)
end-do
```

Similarly for specification of bounds (a bound is just a simple unnamed constraint):

```
forall(i in 1..NI) do
  forall(t in 1..NT) store(i,t) <= MAXSTORE(t)
  store(i,0) = STORE0
end-do
```

May include **conditions** in sums or loops:

```
forall(c in 1..10 | CAP(c)>=100.0)
  MaxCap(c) :=
    sum(i in 1..10, j in 1..10 | i<>j)
      TECH(i,j,c)*x(i,j,c) <= MAXTECH(c)
```

2.3 Index sets

Explicit statement:

```
declarations
  MaxCap: array(1..10) of linctr
end-declarations

forall(d in 1..10)
  MaxCap(d) :=
    sum(p in 1..10, m in 1..10)
      TECH(p,m,d)*x(p,m,d) <= MAXTECH(d)
```

Defining named sets:

```
declarations
  PRODUCTS = 1..5
  MATERIALS = {12,487,163}
  DEPOTS = {"Boston","New York","Atlanta"}

  MaxCap: array(DEPOTS) of linctr
end-declarations

forall(d in DEPOTS)
  MaxCap(d) :=
    sum(p in PRODUCTS, m in MATERIALS)
      TECH(p,m,d)*x(p,m,d) <= MAXTECH(d)
```

Using named sets

- improves the readability of a model
- makes it easier to apply the model to different sized data sets
- makes the model easier to maintain

2.4 Reading data in from text files

Read data into `COST` from `cost.dat`

```
initializations from 'cost.dat'
  COST
end-initializations
```

Data file `cost.dat` (dense data format)

```
COST : [3.9 0 4.8
        0 7.5 5.5]
```

Data file `cost2.dat` (sparse data format)

```
COST: [("Oil1" 1) 3.9  ("Oil1" 3) 4.8
       ("Oil2" 2) 7.5  ("Oil2" 3) 5.5]
```

Mosel data format:

- file may include single line comments, marked with '!'
- format: label, colon, data value(s)
- for an array, use a single list enclosed in [ ]
- list may be comma or space separated
- dense format: the values fill the data table starting at the first position and varying the last index most rapidly
- sparse format: each data item is preceded by the corresponding index tuple (in brackets)

Specifying the absolute path

```
initializations from 'c:/data/cost.dat'
  COST
end-initializations
```

Path relative to current working directory

```
initializations from '../cost.dat'
  COST
end-initializations
```

Read several data tables from a single file

```
initializations from 'cost.dat'
  SCOST
  PCOST
end-initializations
```

Different data label and model object names

```
initializations from 'cost.dat'
  COST as 'COST_DETAILS'
end-initializations
```

Read several data arrays with identical index sets from a single table

```
initializations from 'chess.dat'
  [DUR,WOOD,PROFIT] as 'ChessData'
end-initializations
```

2.5 Writing data out to text files

You can write out values in an analogous way to reading them in

```
initializations to 'cost.dat'
  COST
end-initializations
```

To write out the solution values of variables, or other solution values (slack, activity, dual, reduced cost) you must first put the values into a data table

```
declarations
  make_sol: array(ITEMS,TIME) of real
  obj_sol: real
end-declarations

forall(i in ITEMS, t in TIME)
  make_sol(i,t) := getsol(make(i,t))

obj_sol := getobjval

initializations to 'make.dat'
  make_sol
  obj_sol
end-initializations
```

Alternatively, you can use `evaluation of` directly in the `initializations` block

```
initializations to 'make.dat'
  evaluation of
    array(i in ITEMS, t in TIME) getsol(make(i,t)) as 'make_sol'
  evaluation of getobjval as 'obj_sol'
end-initializations
```

2.6 User defined data formats

Mosel also provides functions which allow you to read data in from and write data out to text files using any format (see list in Section 4.6).

Reading in free format data

```
declarations
  ii, jj: integer     ! Don't use normal i,j
```

```
end-declarations

fopen('cost.dat', F_INPUT)
while(not iseof)
  readln(ii, ',', jj, ',', COST(ii,jj))
fclose(F_INPUT)
```

Writing out data in user format

```
fopen('xsol.dat', F_OUTPUT)
forall(s in SUP, d in DEP)
  writeln(s, ',', d, ',', getsol(x(s,d)))
fclose(F_OUTPUT)
```

2.7 Using other data sources

The `initializations` block can work with many different data sources and formats thanks to the notion of *I/O drivers*.

I/O drivers for physical data files:

- *mmodbc.odbc* for databases with ODBC connector
- *mmsheet.excel* for MS Excel spreadsheets
- *mmsheet.xls* and *mmsheet.xlsx* for generic spreadsheet access, including on non-Windows platforms
- *mmsheet.csv* for CSV format files
- *mmoci.oci* for Oracle databases
- *mmetc.diskdata* for mp-model style data files

Other drivers are available, *e.g.* for data exchange in memory between models or between a model and the host application.

Change of the data source = change of the I/O driver, no other modifications to your model

```
initializations from "mmsheet.xls:mydat.xls"
 COST as 'CostData'
end-initializations

initializations to "mmodbc.odbc:mydat.mdb"
 SOL as 'SolTable'
end-initializations
```

3 Model building style recommendations

- Separation of problem logic and data
  - Typically, the model logic stays constant once developed, with the data changing each run
  - Fix the model and obtain data from their source to avoid editing the model which can create errors, expose intellectual property, and is impractical for industrial size data
- You should aim to build a model with sections in this order
  - *constant data*: declare, initialize
  - *all non-constant objects*: declare
  - *variable data*: initialize / input / calculate

  - *decision variables*: create, specify bounds
  - *constraints*: declare, specify
  - *objective*: declare, specify, optimize

- Use a **naming convention** that distinguishes between different model object types, for example
  - known values (data) using upper case
  - unknown values (variables) using lower case
  - constraints using mixed case
- Variables are *actions* that your model will prescribe
  - Use verbs for the names of variables. This emphasizes that variables represent '*what to do*' decisions
- Try to include 'min' or 'max' in the name of your objective function; an objective function called 'OBJ' is not very helpful when taken out of context!
- Indices are the *objects* that the actions are performed on
  - Use nouns for the names of indices
- Declare all objects in your model (optional unless using compiler option `noimplicit`)
  - Allows the compiler to detect syntax errors more easily
  - Mosel's guessed declaration doesn't always work
  - A form of rigour and documentation
  - An opportunity for a descriptive comment
- **Comments** are essential for a well written model
  - Always use a comment to explain what each parameter, data table, variable, and constraint is for when you declare it
  - Add extra comments to explain any complex calculation etc
  - Comments in Mosel:

```
declarations
  make: array(1..NP, 1..NT) of mpvar    ! Amount of p produced in time t
  sell: array(1..NP, 1..NT) of mpvar    ! Amount of p sold in time t
end-declarations

(! And here is a multi-line
  comment !)    forall(t in 1..NT) ...
```

4 Mosel Language overview

4.1 Structure of a Mosel model

A Mosel model (text file with extension `.mos`) has the form

```
model model_name

  Compiler directives

  Parameters

  Body

end-model
```

Compiler directives

- Options are specified as a *compiler directive*, at the beginning of the model
- Options include `explterm`, which means that each statement must end with a semi-colon, and `noimplicit`, which forces all objects to be declared

```
options explterm
options noimplicit
```

- `uses` statements are also compiler directives

```
uses "mmxprs", "mmodbc"
```

- Can define a version number for your model

```
version 1.0.0
```

Run-time parameters

- Scalars (of type `integer`, `real`, `boolean`, or `string`) with a specified default value
- Their value may be reset when executing the model
- Use `initializations from` for inputting structured data (arrays, sets,...)
- At most one `parameters` block per model

Model body

- Model statements other than compiler directives and parameters, including any number of
  - `declarations`
  - `initializations from` / `initializations to`
  - functions and procedures

Implicit declaration

- Mosel does *not* require all objects to be declared
- Simple objects can be used without declaring them, if their type is obvious
- Use the `noimplicit` option to force all objects to be declared before using them (see item *Compiler directives* above)

Mosel statements

- Can extend over several lines and use spaces
- However, a line break acts as an expression terminator
- To continue an expression, it must be cut after a symbol that implies continuation (*e.g.* `+ - , *`)

4.2 Data structures

`array`, `set`, `list`, `record` and any combinations thereof, *e.g.*,

```
S: set of list of integer
A: array(range) of set of real
```

Arrays *Array*: collection of labeled objects of a given type where the label of an array entry is defined by its index tuple

```
declarations
  A: array(1..5) of real
  B: array(range, set of string) of integer
  x: array(1..10) of mpvar
  C: array(1..5) of real
end-declarations

A:: [4.5, 2.3, 7, 1.5, 10]
A(2):= 1.2
B:: (2..4,["ABC", "DE", "KLM"])[15,100,90,60,40,15,10,1,30]
C:= array(i in 1..5) x(i).sol
```

Sets *Set*: collection of objects of the same type without establishing an order among them (as opposed to arrays and lists)
Set elements are unique: if the same element is added twice the set still only contains it once.

```
declarations
  S: set of string
  R: range
end-declarations

S:= {"A", "B", "C", "D"}
R:= 1..10
```

Lists *List*: collection of objects of the same type
A list may contain the same element several times. The order of the list elements is specified by construction.

```
declarations
  L: list of integer
  M: array(range) of list of string
end-declarations

L:= [1,2,3,4,5]
M:: (2..4)[['A','B','C'], ['D','E'], ['F','G','H','I']]
```

Records *Record*: finite collection of objects of any type
Each component of a record is called a *field* and is characterized by its name and its type.

```
declarations
  ARC: array(ARCSET:range) of record
    Source,Sink: string      ! Source and sink of arc
    Cost: real               ! Cost coefficient
  end-record
end-declarations

ARC(1).Source:= "B"
ARC(3).Cost:= 1.5
```

User types User types are treated in the same way as the predefined types of the Mosel language. New types are defined in `declarations` blocks by specifying a type name, followed by =, and the definition of the type.

```
declarations
  myreal = real
  myarray = array(1..10) of myreal
  COST: myarray
end-declarations
```

4.3 Selection statements

if ... end-if

```
if c=1 then
  writeln('c equals 1')
end-if
```

if ... else ... end-if

```
if c=1 then
  writeln('c equals 1')
else
  writeln('c does not equal 1')
end-if
```

if ... elif ... else ... end-if

```
if c=1 then
  writeln('c equals 1')
elif c>1 then
  writeln('c is bigger than 1')
else
  writeln('c is smaller than 1')
end-if
```

case ... end-case

```
case c of
  1,2 : writeln('c equals 1 or 2')
  3   : writeln('c equals 3')
  4..6: do
          writeln('c is in 4..6')
          writeln('c is not 1, 2 or 3')
        end-do
else
  writeln('c is not in 1..6')
end-case
```

4.4 Loops

forall

```
forall(f in FAC, t in TIME)
  make(f,t) <= MAXCAP(f,t)

forall(t in TIME) do
  use(t) <= MAXUSE(t)
  buy(t) <= MAXBUY(t)
end-do
```

with

equivalent to a `forall` loop stopped after the first iteration

```
with f='F1', t=1 do
  make(f,t) <= MAXCAP(f,t)
end-do
```

while

```
i := 1
while (i <= 10) do
  write(' ', i)
  i += 1
end-do
```

repeat ... until

```
i := 1
repeat
  write(' ', i)
  i += 1
until (i > 10)
```

break, next

- `break` jumps out of the current loop

- `break n` jumps out of *n* nested loops (where *n* is a positive integer)
- `next` jumps to the beginning of the next iteration of the current loop
- use `break 'looplabel'` and `next 'looplabel'` with labeled loops:

```
'L1': repeat
  'L2': while  (condition1) do
    if  condition2 then
      break 'L1'
    end-if
  end-do
until  condition3
```

counter

- Use the construct `as counter` to specify a counter variable in a bounded loop (*i.e.*, `forall` or aggregate operators such as `sum`). At each iteration, the counter is incremented

```
cnt:=0.0
writeln("Average of odd numbers in 1..10: ",
        (sum(cnt as counter, i in 1..10 | isodd(i)) i) / cnt)
```

4.5 Operators

Assignment operators

```
i := 10
i += 20        ! Same as i := i + 20
i -=  5        ! Same as i := i - 5
```

Assignment operators with linear constraints

```
C := 5*x + 2*y <= 20
D := C + 7*y
```

then `D` is

```
D := 5*x + 9*y - 20
```

The constraint type is dropped with `:=`

```
C := 5*x + 2*y <= 20
C += 7*y
```

then `C` is

```
C := 5*x + 9*y <= 20
```

The constraint type is retained with +=, -=

Arithmetic operators

standard:	`+ - * /`
power:	`^`
int. division/remainder:	`mod div`
sum:	`sum(i in 1..10) ...`
product:	`prod(i in 1..10) ...`
minimum/maximum:	`min(i in 1..10) ...`
count:	`count(i in 1..10 | isodd(i))`

Linear and non-linear expressions

Decision variables can be combined into linear or non-linear expressions using the arithmetic operators

- module *mmxprs* only works with linear constraints, so no `prod`, `min`, `max`, ...
- other solver modules, *e.g.*, *mmquad*, *mmnl*, *mmxnlp*, also accept (certain) non-linear expressions

Logical operators

constants:	`true, false`
standard:	`and, or, not`
AND:	`and(i in 1..10) ...`
OR:	`or(i in 1..10) ...`
comparison:	`<, >, =, <>, <=, >=`

Set operators

constants:	`{'A', 'B'}`
union:	`+`
union:	`union(i in 1..10) ...`
intersection:	`*`
intersection:	`inter(i in 1..10) ...`
difference:	`-`

Set comparison operators

subset:	`Set1 <= Set2`
superset:	`Set1 >= Set2`
equals:	`Set1 = Set2`
not equals:	`Set1 <>Set2`
element of:	`"Oil5" in Set1`
not element of:	`"Oil5" not in Set1`

List operators

constants:	`[1, 2, 3]`
concatenation:	`+, sum`
truncation:	`-`
equals:	`L1 = L2`
not equals:	`L1 <>L2`

4.6 Built in functions and procedures

The following is a list of built in functions and procedures of the Mosel language (excluding modules). Functions return a value; procedures do not.

Dynamic array handling	`create`	`exists`	`delcell`	`isdynamic`
Freeze (finalize) a dynamic set	`finalize`			
Rounding functions	`ceil`	`floor`	`round`	`abs`
Mathematical functions	`exp`	`log`	`ln`	`sqrt`
	`cos`	`sin`	`arctan`	
	`isodd`			
Special real values	`isfinite`	`isinf`	`isnan`	
Random number generator	`random`	`setrandseed`		

Minimum/maximum of a list of values	`v := minlist(5, 7, 2, 9)` `w := maxlist(CAP(1), CAP(2))`

Inline "if" function	`MAX_INVEN(t) := if(t < MAX_TIME, 1000, 0)`
	
	`Inven(t) := stock(t) = buy(t) - sell(t) +`
	`if(t > 1, stock(t-1), 0)`

Matrix export to file	`exportprob`

File handling	`fopen` `fclose` `fselect`
	`getfid` `getfname` `getreadcnt`
	`iseof` `fflush` `fskipline`
	`fwrite[_] / fwriteln[_]`
	`read / readln` `write[_] / writeln[_]`

String handling	`strfmt` `substr` `_`

Access and modify model objects	`getcoeff[s]` `setcoeff` `getvars`
	`sethidden` `ishidden` `setname`
	`gettype` `settype` `getsize`
	`makesos1` `makesos2`
	`getfirst` `getlast` `findfirst`
	`findlast` `reverse` `getreverse`
	`gethead` `gettail` `cuthead`
	`cuttail` `splithead` `splittail`

Access solution values	`getobjval`
	`getsol` `getrcost`
	`getslack` `getact` `getdual`

Exit from a model	`exit`
Mosel controls	`getparam` `setparam`
Date/time	`currentdate` `currenttime` `timestamp`
Bit value handling	`bitflip` `bitneg` `bitset` `bitshift` `bittest` `bitval`

Miscellaneous	`asproc` `assert` `reset`
	`setioerr` `setmatherr`
	`publish` `unpublish`

- **Overloading of subroutines**
  - Some functions or procedures are *overloaded*: a single subroutine can be called with different types and numbers of arguments
- **Additional subroutines** are provided by *Mosel library modules*, which extend the basic Mosel language, *e.g.*,
  - *mmxprs*: Xpress Optimizer
  - *mmodbc*: ODBC data connection
  - *mmsystem*: system calls; text handling
  - *mmjobs*: handling multiple models and (remote) Mosel instances
  - *mmsvg*: graphics

  ⇒ See the 'Mosel Language Reference Manual' for full details
- **User-defined functions and procedures**
  - You can also write your own functions and procedures within a Mosel model
  - Structure of subroutines is similar to a `model` (may have `declarations` blocks)

- User subroutines may define overloaded versions of built in subroutines

⇒ See examples in the 'Mosel User Guide' (Chapter *Functions and procedures*)

- **Packages**
  - Additional subroutines may also be provided through *packages* (Mosel libraries written in the Mosel language as opposed to Mosel modules that are implemented in C)

⇒ See the 'Mosel User Guide' for further detail (Chapter *Packages*)

4.7 Constraint handling

```
Ctr1:= 2*x + y <= 10         ! Named constraints
Ctr2:= x is_integer

2*x + y <= 10                ! Anonymous constraints
y >= 5
```

Named constraints can be	accessed:	`val:= getact(Ctr)` `getvars(Ctr, vars)`
	hidden:	`sethidden(Ctr, true)`
	redefined:	`Ctr:= x+y <= 10` `Ctr:= 2*x+5*y >= 5`
	modified:	`Ctr += 2*x` `settype(Ctr, CT_UNB)`
	deleted (reset):	`Ctr:= 0`

Anonymous constraints are constraints that are specified without assigning them to a `linctr` variable. *Bounds* are (to Mosel) just simple constraints without a name. Anonymous constraints are applied in the optimization problem just like ordinary constraints. The only difference is that it is not possible to refer to them again, either to modify them, or to examine their solution value.

4.8 Problem handling

- Mosel can handle several *problems* in a given *model* file. A default problem is associated with every model.
- Built in type `mpproblem` to identify mathematical programming problems
  - The same decision variable (type `mpvar`) may be used in several problems
  - Constraints (type `linctr`) belong to the problem where they are defined
- The statement `with` allows to open a problem (= select the active problem):

```
declarations
  myprob: mpproblem
end-declarations
...
with myprob do
  x+y >= 0
end-do
```

- Modules can define other specific problem types. New problem types can also be defined by combining existing ones, for instance:

```
mypbtyp = mpproblem and somepbtype
```

- Problem types support assignment: `P1:= P2`
  and additive assignment: `P1 += P2`

4.9 Reserved words

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

```
a:   and  array  as
b:   boolean  break
c:   case   count   counter
d:   declarations   div   do   dynamic
e:   elif   else   end   evaluation
f:   false   forall   forward   from   function
i:   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
l:   linctr   list
m:   max   min   mod   model   mpvar
n:   next   not
o:   of   options   or
p:   package   parameters   procedure   public   prod
r:   range   real   record   repeat   requirements   return
s:   set   string   sum
t:   then   to   true
u:   union   until   uses
v:   version
w:   while   with
```

4.10 Annotations

- *Annotations* are meta data in a Mosel source file that are stored in the resulting BIM file after compilation; no impact on the model itself (treated like comments); either global or associated with public globally declared objects (including subroutines).
- Single-line annotations start with '!@' and a name; blocks are surrounded by '(!@' and '!)'
- !@doc.descr denotes the annotation marker descr within category doc (predefined category names are mc and doc, user-defined names can also be employed)

```
(!@doc.   Enter category 'doc' (this text is ignored)
 @ descr  This is the value of 'doc.descr'
 @.       Jump back to root (this text is ignored)
 @mynote  Contents of 'mynote' (full name: '.mynote')
 @.anote  Complete form of an annotation in default category
!)
```

- *Declaring annotations* (via the mc.def compiler annotation): optional; enables the compiler to check the validity of the definitions and reject non-compliant ones

```
! Defining an alias that redirects onto 2 different annotations:
!@mc.def descr alias doc.descr om.descr
```

- *moseldoc* tool: generates an XML model documentation that is processed into HTML pages

  1. Compile source model file with option -D

     ```
     mosel comp -D mymodel.mos
     ```

  2. Run program moseldoc

     ```
     moseldoc mymodel                  Generates HTML and XML
     moseldoc -o mydir -html mymodel   HTML only, specifying output directory
     moseldoc -f -xml mymodel          XML only, forcing output overwrite
     ```

  See 'Mosel Language Reference', section *Documenting models using annotations* for a list of the doc annotations

5 Using the Mosel Command Line

The Mosel Command Line is supported on all platforms that Mosel can be run on.

Standard sequence for model execution from the command line:

```
mosel exec mymodel.mos           Execute (=compile/load/run) model 'mymodel.mos'
mosel mymodel                    Short form (works with 'mymodel.mos' or 'mymodel.bim')
```

Some useful commands (see 'Mosel Language Reference manual' for the full list):

Command line help text:	`mosel -h`
Mosel version:	`mosel -V`
Display functionality:	`mosel exam[ine] [-cspthirvaum]`
Execute a model file:	`mosel exec[ute]`
Compile a model file:	`mosel comp[ile] mymodel.mos`
Load and run a BIM file:	`mosel run mymodel.bim`
Start the debugger:	`mosel debug mymodel.mos`
Run the profiler:	`mosel prof[ile] mymodel.mos`
Perfom a code coverage run:	`mosel cover[age] mymodel.mos`
List available modules/packages:	`mosel lslib`

Examples:

```
mosel comp mymodel.mos -o mybim.bim   Compile to a specified BIM file name/location
mosel prof mymodel.mos           Perform a profiler run (output in 'mymodel.mos.prof')
mosel exam -h                    Display Mosel version info and paths
mosel exam -a mybim.bim          Display annotations of a model or package
mosel exam -ps mmxprs            Display parameters and subroutines of module 'mmxprs'
```

Setting model runtime parameters:

```
mosel exec mymodel NT=5 DATAFILE="mynewdata.dat"    Source in mymodel.mos
mosel run mymodel NT=5 DATAFILE="mynewdata.dat"     Loads mymodel.bim
mosel mymodel NT=5 DATAFILE="mynewdata.dat"         With mymodel.mos or mymodel.bim
```

5.1 Debugger commands

Breakpoints:	`break  delete  bcond[ition] breakpoints breaksub`
Model execution:	`cont  next  step  finish  model`
Output:	`display undisplay list print info exportprob` `lsattr lslibs lslocal lsmods lssymb`
Stack access:	`up  down  where`
Interpreter options:	`option`
Termination:	`quit`

Example: Simple debugging sequence

```
mosel debug debugexpl.mos        Start Mosel debugger
break 20                         Set breakpoint at line 20
cont                             Execute up to the breakpoint
print D                          Print out symbol 'D'
cont                             Continue model execution
info Arr                         Information about model object 'Arr' (e.g. size)
lsmods                           Display model info (e.g. memory usage)
quit                             Quit the debugger
```

Example: Debugging a submodel

```
mosel debug debugmaster.mos      Start Mosel debugger
breaksub 1                       Stop at start of submodels
cont                             Execute up to the breakpoint
break 25 debugsub.mos            Set breakpoint in the submodel
display SNumbers                 Display watch on object 'SNumbers'
cont                             Execute up to the breakpoint
break 31 debugsub.mos            Another submodel breakpoint
bcond 2-2 SNumbers.size < 10     Condition on 2nd submodel breakpoint
cont                             Execute up to the breakpoint 2-2
quit                             Quit the debugger
```

6 Working with Xpress Workbench

Xpress Workbench is a graphical development environment for Mosel models and Xpress Insight applications.

Workbench panes

Model editor (central window), project directory navigation and command history (left), model output and execution log information (bottom), debugging, deployment and collaboration information (right).

Workspace preferences (settings).
Toggle full-screen view for logging pane.

Use menu *Window*≫*Presets*≫*Full IDE* to restore original window layout.

Editor

Code folding and breakpoint markers appear in the grey area immediately left to the text.

Open a new file/tab
Subdivide and re-arrange panes in the editor window
Code folding for blocks of Mosel statements
Unfold folded code
Line position markers during debugging

Model execution

The name of the model is selected in the box next to these buttons, it may be different from the model(s) opened in the editor.

Execute (compile/load/run) a model.
Execute a model in debug mode.
Start a tuning run for an optimization problem.

Use menu *Run*≫*Build* to compile a model.

Debugger

Breakpoints are set by clicking onto the gray area (left to the line number if it is displayed) preceding each row in the editor window, breakpoint conditions can be added via the right mouse button menu on the breakpoint icon.

Delete breakpoint/desactivated breakpoint.
Delete a conditional/desactivated conditional breakpoint.

Navigating in the debugger:

Activate/desactivate all breakpoints.
Start/stop the debugger.
Resume/suspend model execution.
Step over an expression.
Step into an expression.
Step out of an expression.
Don't pause on exceptions.

Deployment to Xpress Insight

- Publish selected model to Insight.
- Build an Insight app archive.
- Debug a scenario.
- Edit Tableau workbooks.
- Refresh Insight scenario tree.