Xpress-Mosel User Guide

Release 2.0

目次

第 1 章

はじめに

1.1 概要

Xpress-Mosel は，Xpress-MP ソフトウエアパッケージの中の初歩的なモデリングコンの構成要素です．これは，実世界の問題を解くことのできるモデルとして定式化する数理計画法の解法の開発において，利用者にたいへん用途の多い環境を提供します．Mosel は，モデリングとプログラミングの言語を組み合わせたもので，コンパイル可能であり，データを集め，解法の情報にアクセスしながらモデルを走らせます．これは，ワンストッププログラミング環境を提供する Xpress-Modeler と Xpress-EMOSL の能力を凌駕するものです．更に，Xpress-Mosel は，他の特別なソルバーエンジンと独立であり，産業用の標準出力形式をサポートしていて，現在入手することができる様々な最適化プログラムと互換性があります．

いろいろな ODBC インタフェースがサポートされているので，Mosel ユーザーは，Xpress-MP ソフトウェアと，業界標準をサポートする多くの一般的なスプレッドシートやデータベースプログラムの間のシームレスな対話を楽しむことができます．このような，Mosel の処理後のインタフェースによって，モデルが自然なデータの形式で得られ，また，解法の詳細は既製のグラフィックディスプレイのために最適化されます．様々な異なるインタフェースを経てアクセス可能な Mosel の能力は，多種多様な一般的なプラットフォーム上で全ての環境におけるユーザーが楽しむことができます．

Mosel アプリケーションは，Xpress-MP の統合されたビジュアル環境である Xpress-IVE を経由してアクセスするのがおそらく最も簡単です．一般的なグラフィックインタフェースを使えば，Mosel の処理後の全ての側面への単純なアクセスを提供しながらモデルを開発，解決することができます．そのグラフィックインタフェースは他のインタフェースによって供給されるよりも，より多くの解法のビジュアルプレゼンテーションをするためグラフィックの拡張機能を備えています．これは，Mosel の新しいユーザーのために，最も簡単なエントリポイントを提供します．

Mosel は，スタンドアロンバージョンでは，一般的なコマンドを実行するために，バッチモード，または，コマンドラインインタープリタによってシンプルなインタフェースを提供します．この ”コンソールモード ”は，モデルを解決したり解法の様々な側面にアクセスしたり，行列ファイルを見たり保存したりするための，多くの機能を提供します．強力なモデル管理システムによって，いくつかのモデルは同時にメモリに保持されて，必要に応じて使うことができます．

Mosel は，ライブラリユーザーのために，作業のモデル化と解法の全ての範囲を実現するための，ルーチンの包括的なセットを提供します．解法にアクセスして行列表現を操作することによってモデルを読み込

んだり解決したりするばかりではなく，動的なモジュールの直接的な扱いや同時に実行される多重モデルの管理をすることもできます．Mosel ライブラリは，作業に応じたいくつかの異なる言語で利用可能です．Xpress-MP ライブラリは，C 言語，Java，Visual Basic をサポートしており，利用者のプログラミングのバックグラウンド，好みに応じて，利用者に適した方法でアクセスすることができます．

このマニュアルは，Mosel に含まれる全ての多くの特徴のリファレンスを提供します．新しいユーザーにとっては，不必要に重いと思うかもしれません．そのようなユーザーは，付随の Xpress-MP Essentials ガイドを見てください．これは，Xpress-MP，及び，そのインタフェースへのやさしい導入になっています．

1.2 Mosel とは

Mosel は，問題をモデリングと解法のための環境です．モデル化言語でもあり，プログラミング言語でもある言語を提供します．Mosel 言語のオリジナリティは，モデリングの文（例えば，決定変数の宣言や定数の表現）と実際に問題を解く手続き（例えば，最適化コマンドの呼び出し）が一体となっているところです．これによって，モデリングと解法の文を組み合わせた複雑な解法のアルゴリズムをプログラムすることができます．

問題の各カテゴリは変数の型と制約に由来し，1 種類のソルバーは全てのケースに関して有能であるとは限りません．これを考慮するために，Mosel システムは，デフォルトではソルバーを統合せずに，モジュールとして供給された外部のソルバーに動的なインタフェースを提供します．各ソルバーモジュールは，Mosel 言語の語彙と能力を直接拡張する手続きと関数の集合から成ります．Mosel と解法のモジュールの間のリンクはメモリレベルで行われ，コアシステムの修正を全く必要としません．

このオープンアーキテクチャは，Mosel を他のソフトウェアとつなぐ方法としても使うことができます．例えば，モジュールは特定のデータベースと通信するのに必要とされる機能を定義します．

モデリングと解法の作業は，通常ソフトウェアアプリケーションによって行われる唯一のオペレーションではありません．したがって，Mosel 環境では，ライブラリとスタンドアロンプログラムの両方の形が提供されます．

1.3 一般的な構成

Mosel への入力は，実行するためのモデル/プログラムのソースを含むテキストファイルです．このソースファイルはまず，Mosel コンパイラによってコンパイルされます．この処理の間，モデル/プログラムの文法がチェックされますが，オペレーションは実行されません．コンパイルの結果は BInary Model(BIM) であり，第 2 のファイルに保存されます．この形式では，モデル/プログラムは実行される準備が整っており，ソースファイルは必要ありません．実際にモデル/プログラムを「実行する」ときには，BIM ファイルが再び Mosel によって読み込まれて，その後，実行されます．これらのことは，Mosel 環境を構成する異なるモジュールによって処理されます．

ランタイムライブラリ: このライブラリは，VIrtual MAchine (VIMA) インタープリタを含みます．どのようにバイナリ形式においてモデル/ プログラムをロードし，どのように実行するかを知っているライブラリです．また，(一度にいくつかのモデルを扱うための) モデルマネージャ，(あるモデルによって必要とされるモジュールをロードしたりアンロードするための) 動的共有オブジェクトマネージャの実行も行います．このライブラリの全ての機能は，ユーザーアプリケーションからアクセスすることができます．

コンパイラライブラリ: このモジュールの役割は，ソースファイルを VIMA インタープリタが実行することができるバイナリ形式に変換することです.

スタンドアロンアプリケーション: 'mosel ' アプリケーションは，上の 2 つのモジュールとリンクされたコマンドラインインタープリタです．モデルをコンパイルして実行するためのプログラムを供給します.

様々なモジュール: これらのモジュールは，最適化の手続きのような Mosel セットに含まれる様々な機能を提供します．例えば,「`mmxprs`」モジュールによって Mosel 言語が拡張され，Xpress-Optimizer を使って現在の問題を最適化する手続き「`maximize`」を利用できるようになります．このようなモジュール化された構造に様々な利点があります.

- モデルはいったんコンパイルされれば，再コンパイルする必要性なしで何回も，例えば異なるデータセットについて，実行することができます.
- コンパイルされたプログラムは，システム，アーキテクチャとは独立です．Mosel ランタイムライブラリと必要なモジュールが装備されていれば，あらゆるオペレーティングシステム上で実行することができます.
- BIM ファイルは，全く記号を含まないで生成することができます．バイナリ形式でモデルを配布しても，知的財産という観点から安全です.
- Mosel は，ライブラリとして更に大きいアプリケーションに容易に統合することができます．モデル/プログラムは，ランタイムライブラリとリンクされた BIM ファイルとアプリケーションとして供給することもできます.
- Mosel システムは，いかなる種類のソルバーも統合しませんが，モジュールが解法の機能を与える方法で設計されています．これにより，Mosel が異なるソルバーとリンクされ，直接メモリを通じてそれらと通信することが可能になっています.
- Mosel はオープンアーキテクチャであるため，Mosel の内容を修正することなく，ケースバイケースで機能の拡張が可能です.

1.4 リファレンス

Mosel は，2，3 のよく知られた技術の，オリジナルなコンビネーションであるということができます．以下に，Mosel の最も重要な「オリジナル」な点をいくつか列挙します.

- システムのアーキテクチャ全体 (コンパイラ，バーチャル・マシン，固有のインタフェース) は，Java 言語によって直接呼ばれます．このようなインプリメントの方法は，人工知能のための言語 (例えば Prolog，Lisp) で一般的に使われる方法です.
- Mosel 言語の文法と主要なブロックは，単純化されているという側面もありますが，一部は Pascal 言語の拡張というという側面もあります.
- 総和演算子 (「`sum`」など) は,「モデルビルダーの伝統」を継承しており，今日の大部分のモデルリング言語と同じです.

- 動的な配列と，その集合との特殊なリンクはおそらく Mosel 特有のものですが，それは，mpmodel モデルビルダーの疎なテーブルの一般化を基にしています.

1.5 本マニュアルの構成

このマニュアルは 2 つのパートに系統立てることができます．第 2 章「Mosel 言語」では，Mosel のモデル化，及び，プログラミング言語を構成する基本的なブロックについて，どのように拡張することができるかを示しながら述べます．続く第 3 章「Mosel の実行」では，Mosel モデルファイルを作成することができる様々なインタフェース，及び，アプリケーションの概観を示します．この章ではコンソールとライブラリコマンド両方について述べますが，どちらのインタフェースのユーザーにとっても有益でしょう.

第 4 章「Mosel 関数と手続き」では，現在 Mosel 言語に定義されている全ての関数と手続き関数について記述します．第 5 章「コンソール，ライブラリファンクション」は，コンソールとライブラリユーザーの両方にとって機能的なリファレンスです．次の 2 つの章、第 6 章「制御パラメータ」，第 7 章「問題の属性」では，Mosel モデルが利用可能な様々な制御や属性を示します.

1.6 表記法

このマニュアルは標準的な印刷の慣例に従っています．コンピュータのコードは `fixed width` フォント，数式や変数は *italic* フォントです．ライブラリ関数では，配列等の構造のために C 言語のスタイルが使われます．コンソールモードルーチンは大文字で表示します．コマンドラインインタープリタは大文字か小文字かを区別しませんが，実際上は小文字が使われます．コンソールとライブラリの両方で等価なコンソールモードのルーチンは，コンソールコマンドを括弧で囲んで，通常，大文字，小文字の両方を表示します．コンソールユーザーには無関係である節では，ライブラリ形式のみが表示されます．コマンドとコントロールの，コンソールとライブラリでの文法の違いについては，第 5 章「コンソールとライブラリ関数」に書かれています．以下のようなルールで記述されています.

- 大括弧 [...] に囲まれたものはオプションを表します.
- 中括弧 {...} に囲まれたものはオプションを表しますが，囲まれたもののうちどれかひとつを選ばなくてはいけません.
- *italic* で書かれたものは，メタ変数を表す表現です．メタ変数に続く記述は，どのように使うかを表しています.
- 「return」は「キーボードでリターンキーかエンターキーを押す」ことを示します.
- CTRL が文字の前に書かれていたら,「CTRL キーを押したまま次に書いてある文字を押す」ことを示します．コントロールキーは，キーボードの左下か角にあります.
- 垂直線「|」はスクリーン上では中央に隙間があるように表示されることもあります．UNIX ではパイプを表すのに使われます．この記号は，PC のスクリーンのドローボックスで使われる記号とは違うことに注意してください．ASCII では,「|」は 16 進で 7C，8 進では 124 です.

第 2 章

Mosel言語

Mosel 言語は，モデリングのための言語であると考えることもできるし，また，プログラミングのための言語であるとも考えることができます．他のモデリングのための言語同様，問題，決定変数，制約，集合や配列といった様々なデータ型やデータ構造を宣言し操作するために必要な機能を提供します． 一方，プログラミング言語に適したいろいろな機能も提供します．すなわち，コンパイル，最適化，全ての便利な制御構造の構成概念（選択，ループ）がサポートされており，それらはモジュールとして拡張可能です．これらの拡張のうち，オプティマイザーは他のモジュールや提供されている機能のようにロードすることができ，他の Mosel の手続きや関数と同様に使うことができます．これらの性質によって，Mosel は，複雑な解法アルゴリズムを記述することのできる，パワフルなモデリング，プログラミング，そして解法のための言語になっています．

文法は学びやすく，またメンテナンスしやすいように設計されています．その結果として，予約語と文法構成の集合は意図的に小さくされ，よくモデリング言語に用意されているようなショートカットやトリックは避けてあります．ショートカットやトリックのような機能はモデルのソースを読みやすくするというより，むしろサイズを小さくするのに役立つことがありますが，理解したりメンテナンスするのを難しくするような，言語自身の矛盾や不確かさを生むこともあるのです．厳密さの主な利点は，ルールが確立されたら，それは言語の全てのところで正しいということです．例えば，どこで集合を使おうとした場合にも，あらゆる種類の集合の表現を使うことができます．

2.1 はじめに

コメント

コメントは，ソースファイルの一部ですが，コンパイラには無視されます．通常は，プログラムが何をしようとしているかを説明するのに使います．ソースファイルの中では，一行のコメントも複数行のコメントも，両方使うことができます．一行の場合には，コメントは「！」で始まり，行の最後で終わります．複数行の場合には，コメントは「(！」と「！)」で囲む必要があります．いくつかの複数行のコメントをネストすることができることに注意してください．

```
!コメントの中
この行は解析される
```

```
(! 複数行のコメントの始まり
  (! 違うコメント
      コメント
  第 2 のレベルのコメント終わり!)
最初のレベルのコメント終わり!) この部分から解析される
```

コメントはソースファイルのどこでも入れることができます.

識別子

識別子はオブジェクトに名前をつけるために使われます. 識別子はアルファベット, 数字,「__」から成る文字列で, 英字か「__」で始まります. 識別子の全ての文字はについて, 大文字小文字が区別されます. (識別子「word」は「Word」と等価ではありません.)

予約語

予約語は特別な意味を持つ識別子で, 言語の中で特別な役割を決定します. この特別な役割によって, これらの「キーワード」はオブジェクトを定義するために使うことはできません. (つまり, 再定義することはできません.) 予約語のリストは次の通りです.

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

Mosel の辞書的アナライザは大文字小文字を区別しますが, 予約語は大文字・小文字の両方で定義されていることに注意してください. (すなわち,「AND」も「and」も予約語ですが,「And」は違います.)

命令の分割, 行の中断

ソースコードを読みやすくするために, それぞれの文は複数の行に分けることができ, 必要であれば空白やタブを使ってインデントすることができます. しかし, 行の中断は文の終わりを意味するので, 文を分割するときには, アナライザに式に, 文が後の行に続いていることを教えるために, 演算子 (「+」「−」..) やカンマ (「,」) のように, 文が続くことをあらわす記号の後で切るようにしなければなりません.

```
A+B  !式 1
-C+D !式 2
A+B- !式 3 ...
C+D  !式 3 の終わり
```

また,「;」という文字は式の終わりを意味するのに使います.

`A+B ; -C+D !2` つの式が同じ行にある

特別な記号でそれぞれの式の終わりを明示的に示すことを好むユーザもいますが，これは，コンパイラのデフォルトの動作を無効にするオプション `explterm`(2.3 節参照) を使えば可能です．このような場合には，行の中断は式の分割であるとはみなされなくなるので，それぞれの文の終わりに「;」という記号をつけて終わらせなければなりません．

```
A+B;  !式 1
-C+D; !式 2
A+B   !式 3 ...
-C+D; !式 3 の終わり
```

この文章の約束

以後の節では，言語の文法を説明します．全てのコードのテンプレートにおいて，以下のような約束をします．

- `word`：「word」はキーワードであり，この通りに書かなければいけません．
- *todo*：「todo」は後で説明されるほかのものに置き換えられます．
- [*something*]：「something」はオプションで，命令のブロック全体を省略可能です．
- {*something...*}：「something」はオプションで，何度も繰り返して使うことができます．

2.2 ソースファイルの構成

Mosel のソースの一般的な構成は以下の通りです．

`model` *model_name*
  [*Directives*]
  [*Parameters*]
  [*body*]
`end-model`

「`model`」文はプログラムの始まりを,「`end-model`」文はプログラムの終わりをあらわします．これ以降に書かれたものは全て無視されます.（これにより，平文のコメントをプログラムの最後に書くことができます.)「*model_name*」は，引用符でくくられた文字列か識別子で，Mosel マネージャの中でモデルの名前として使われます．これに続く「*directives*」と「*parameters*」はオプションです．実際のプログラム/モデルは,「*body*」の部分に書かれ，宣言ブロック，サブルーチンの定義と文の順番で並んで構成されます．この言語は,「手続型」であって,「宣言型」ではないということを理解することが重要です．つまり，宣言と文は，現れた順番にコンパイルされ，実行されます．その結果として，ソースファイルの中で後で宣言された識別子を参照することはできないし，すでに実行されたソースファイルの文を後で考えることもできません．その上，言語は「コンパイルされる」のであって,「インタープリットされる」わけではありません．

つまり，ソースファイル全体がまずはじめにバイナリ形式（「BIM ファイル」）に翻訳され，その次に，このバイナリ形式のプログラムが実行されるために読まれます．コンパイル中は，いくつかのシンプルな定数式を除いて，何の動作もされません．これが，コンパイル中にほんの少しのエラーしか検出できず，多くのエラーは実行時に検出される理由です．

2.3 コンパイラ命令

コンパイラは,「uses」文と「options」文という，2 つのタイプの命令を受け付けます．
一般的な「uses」文は以下のような形式をとります．

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

これは，コンパイラに列挙したモジュールを読み込んで，それらが定義する記号をインポートさせます．
コンパイラのオプションは，コンパイラのデフォルトの動作を変えるために使います．
一般的な「options」文は以下のような形式をとります．

options *optname1* [,*optname2*...]

サポートされているオプションは次の通りです．

- expl term これは，式の終わりを明示的に与えるためのオプションです．(「命令の分割，行の中断」参照)
- noimplicit これは，暗示的な宣言を使わないためのオプションです．(「暗示的な宣言について」参照)

例えば，以下のように使います．

```
uses "mmsystem"
options noimplicit,explterm
```

2.4 パラメータブロック

モデルパラメータは記号で，モデルを走らせる前に値をセットすることができます (XPRMrunmod(RUN) コマンドのオプションパラメータ)．一般的なパラメータブロックは以下のような形式をとります．

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

ここで，各識別子 *identi* はパラメータの名前で，対応する式 *Expressioni* はそのデフォルトの値です．プログラムのはじめに明示的な値の代入がない場合にはこの値が代入されます．(例えば，XPRMrummod(RUN) コマンドのパラメータ）モデルパラメータはソースファイルのなかでこの後ずっと定数として扱われます．(オリジナルの値を変えることはできません.)

```
parameters
  size=12
  F="myfile"
end-parameters
```

2.5 ソースファイルの包含

Mosel プログラムは，ファイルを包含することにより，いくつかのソースファイルに分割することができます．include 命令は次のように使います．

include *filename*

ここで，*filename* は包含されるファイルの名前です．このファイルの名前が拡張子を持たない場合，自動的に「.mos」という拡張子がつきます．

include 命令は，コンパイル時に *filename* という名前のファイルの内容に置き換えられます．

「a.mos」という名前のファイルが下のような内容であるとします．

```
model "Example for file inclusion"
  written ('From the main file')
  include "b"
end-model
```

また,「b.mos」は以下のような内容であるとします．

```
  written ('From an included file')
```

「b.mos」を包含することにより,「a.mos」は以下と等価になります．

```
model "Example for file inclusion"
  written ('From the main file')
  written ('From an included file')
end-model
```

命令のブロックの中や，プログラム本体ではファイルの包含ができないことに注意して下さい．(従って，包含されるファイルでは，uses，options，parameters のような命令は含んではいけません．)

2.6 宣言ブロック

宣言ブロックの役割は，プログラム/モデルが使う処理のパートのエンティティに名前，型，構造を与えることです．値の型はそのドメイン（例えば，整数か実数か）とその構造，つまり，どのように組織され，格納されるか（例えば，1 つの値を参照するのか配列の形を持つ順序つきの集合なのか）で定義されます．宣言ブロックは,「declarations」と「end-declarations」で囲まれた宣言文のリストから成ります．

declarations
  Declare stat

[*Declare stat ...*]
end-declarationss

ひとつのソースファイルにいくつかの宣言ブロックがあることもありますが，あるブロックで導入された記号は，そのブロックより前に使うことはできません．一度なにかのエンティティに代入された名前は他の何にも使うことができません．

エレメンタリー型

エレメンタリーオブジェクトは，集合や配列のような，より複雑なデータ構造を作るために使います．もちろん，エレメンタリー型のひとつの値を参照するものとして，エンティティを宣言することも可能です．そのような宣言は以下の通りです．

ident1 [, ident2 ..] : typename

ここで，*typename* はオブジェクトの型です．各識別子 *identi* は与えられた型の値の参照を宣言します．

型の名前は基本的な型（integer,real,string,boolean）か，MP 型（mpvar,linctr）です．MP 型は，数理計画法に関係し，決定変数と線形制約条件の宣言を許します．線形制約条件オブジェクトは一次式を格納するのにも使えることに注意してください．

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

基本的な型

基本的な型には以下のものがあります．

- integer　-214783648 から 214783648 の値をとる整数値．
- real　$-1.7e+308$ から $1.7e+308$ の値をとる実数値．
- string　テキスト．
- boolean　ブーリアン（論理）式の結果．ブーリアンエンティティの値は，記号「true」か「false」のどちらかをとる．

宣言のあとに，各エンティティは初期値 0，空の文字列，または「false」を受け取ります．

MP 型

数理計画法のために，2 つの特別な型が用意されています．

- mpvar　決定変数
- linctr　線形制約条件

集合

集合は，与えられた型のエレメントを集めたものとして使われます．集合の宣言は，集められるエレメントの型の定義から成ります．

一般的な集合の宣言は以下のような形式をとります．

ident1 [,*ident2*....]:`set of` *typename*

ここで，*typename* は基本的な型のうちのひとつで，各識別子 *identi* はその型の集合として宣言されます．

一般的な形式より便利であるために好まれることがある，特別な集合の型もどこでも使うことができます．範囲集合は，与えられた間隔で連続した整数の集まりです．範囲集合の宣言は以下のようにして行います．

ident1 [, *ident2*....]:`range set of integer`

各識別子 *identi* は整数の範囲集合として宣言されます．

全ての新しく作られた集合は空です．

```
declarations
  sl: set of string
  rl: range
end-declarations
```

配列

配列は与えられた型のラベルつきオブジェクトの集合です．ラベルは，集合によって特徴付けられたドメインにその値をとるような添字のリスト, 添字集合によって定義されます．配列は固定された大きさのものも動的な大きさのものも使うことができます．固定された大きさの配列は，大きさ（つまり，それが含むオブジェクト（セル）の総数）は宣言したときに決まります．全ての必要なセルが作られ，直ちに初期化されます．動的な大きさの配列は，作られたときは空です．セルは値を代入したときに作られ（「代入」参照），配列は「オン・デマンド」で大きくなります．作られていないセルの値は，配列の型のデフォルトの初期値です．一般的な配列の宣言は以下のような形式をとります．

ident1 [, ident2...] : `[dynamic] array` (*list_of_sets*) `of` *typename*

ここで，*list_of_sets* は，カンマで区切られた宣言/式の集合のリストで，*typename* は基本的な型のひとつです．各識別子 *identi* は与えらた型の配列として宣言され，与えられた集合によって添字が決められます．添字集合では，集合の宣言は無名で行えます.（つまり,「`rs:set of real`」は，もし「`rs`」への参照が必要ないなら,「`set of real`」で置き換えることができます.）

```
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
```

デフォルトでは，全ての添字集合のサイズが固定なら配列のサイズは固定です.（つまり，定数か finalize 手続きによって確定されています.）そうでなければ，動的にサイズが変えられます．装飾子「dunamic」は，配列を動的なものに変えるのに使います.

一度集合が添字集合として使われたら，Mosel は，配列のエントリにアクセスできるようすることを保障するために，そのサイズを減らすことはありません．このような集合は固定です.

mpvar 型の動的な配列の特殊なケース

mpvar 型の配列が動的に定義され，宣言したときにその添字集合の少なくともひとつのサイズが未知の場合，対応する変数は作られません．この場合，決定変数に値を代入する方法がないので，配列の関連する各エントリは「create」手続きを使って作る必要があります.

定数

定数は，宣言時に値が既知で，変更されることのない識別子です．一般的な定数の宣言は以下のような形式をとります.

identifier = Expression

ここで，identifier は定数の名前で，Expression はその初期値であり，唯一とる値です．基本的な型のうちのひとつか，それらの型のうちのひとつの集合です.

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

2.7 式

式は，キーワードを伴い，言語を形作る主たるブロックです．この節は，式を作るための基本的な演算子とコネクタをまとめます.

はじめに

式は，定数，演算子,（オブジェクトまたは関数の）識別子を用いて構成します.

識別子が式の中で現れた場合，その値はこの識別子で参照されます．集合か配列の場合は，構造全体です．配列のひとつのセルにアクセスするためには，配列を「再参照」する必要があります．配列の再参照は以下のように行います.

array_ident (Exp1 [, Exp2...])

ここで，*array_ident* は配列の名前で，*Expi* は，配列の *i* 番目の添字集合の型の式です．このような式の型は，配列の型であり，その値は配列に「*Exp1[,Exp2...]*」のようなラベルで格納されます．

関数呼び出しは以下のようになります．

function_ident

または，

function_ident(Exp1 [, Exp2...])

ここで，*function_ident* は関数の名前で，*expi* はこの関数で必要な i 番目のパラメータです．

特別な関数「`if`」を用いると式と式の選択ができます．文法は次の通りです．*Bool_expr* の値が真ならば *Exp1*，そうでない場合は *Exp2* を採用します．

if(Bool_expr, Exp1, Exp2)

この式の型は，*Exp1* と *Exp2* です．*Exp1* と *Exp2* の型は一致している必要があります．

Mosel コンパイラは，以下のような場合，与えられた演算子によって必要な型変換を自動的に行います．

- リストの配列が異なるとき

  ```
  integer -> real
  ```

- 関数と手続きに関するパラメータリスト

  ```
  integer -> real,linctr
  real ->linctr
  mpvar ->linctr
  ```

- 任意の箇所

  ```
  integer -> real,string,linctr
  real ->string,linctr
  mpvar ->linctr
  boolean -> string
  ```

型の名前を関数として用いることによって（すなわち,「`integer`」,「`real`」,「`string`」,「`boolean`」），基本的な型変換を行うことが可能です．string の場合，変換された結果は変換された式の原文どおりの表現になります．boolean の場合，数値結果がゼロでなければ真で，string の場合，結果の string が「true」という単語ならば真です．変換は，MP 型や構造的な型には定義されていないことに注意してください．

`!A=3.5,B=2` であるとする．

```
integer(A+B)        !=5
string(A-B)         !=''1.5''
real(integer(A+B)) !=5.5 （コンパイラが式を簡略化するため）
```

括弧 () は，あらかじめ決められた演算子の評価の順番を変えるため，あるいは，式の集合を簡略化するのに用います．

総和演算子

各数値の集合を定義するような添字のリストに作用する演算子は総和演算子といわれます．この演算子は値の可能なそれぞれの組に対して作用します．

総和演算子の一般的な形式は以下のようになります．

Aggregate_ident (Iterator1 [, Iterator2 ⋯]) Expression

ここで，*Aggregate_indent* は演算子の名前，*Expression* はこの演算子とコンパチブルな式です (下の異なる演算子が使える例参照)．このような総和演算子の結果の型は，*Expression* の型と同じです．

iterator は次のようなコンストラクトのうちのひとつを用いて表されます．

Set_expr

または
ident1 [,ident2 ...]`in` *Set_expr[— Bool_expr]*

または
ident = Expression [— Bool_expr]

最初の形式は，添字名を指定せずに値のリストを与えます．2 番目の形式は，添字名 *identi* が *Set_expr* によって定義される集合の全ての値を連続してとります．3 番目の形式では，添字名 *identi* に，ひとつの値（スカラーでなければならない）が代入されます．2 番目と 3 番目では，作られた識別子のスコープが，演算子のスコープに限定されます．(すなわち，それは続く iterator と総和演算子のオペランドのためのみ存在します.）その上，オプショナルな条件が *Bool_expr* でつけられ，これは添字のドメインの関連のある要素を選ぶためのフィルタになります．この条件は，可能な限り早く評価されることに注意しなければなりません．その結果，ブール式は考慮されている iterator のリストの中に定義されたいかなる添字にもよらず，総和演算子の前に一回だけ評価され，添字の可能なタプルごとのためには評価されのではありません．

添字は定数と考えられます．名前をつけられている添字の値を（例えば代入などを使って）明示的に変えることはできません．

算術式

数値定数は一般的な科学的記述を使って書くことができます．算術式は，自然に普通の演算子 (+，−，∗，/（割り算），単項 −，単項 +，^（べき乗）) を使って書くことができます．整数値では,「`mod`」（割った余り）と「`div`」（整数の割り算）も定義されています．`mpvar` オブジェクトは，式の中で実数値と同様に扱われることに注意してください．

「`sum`」（和）という総和演算子は integer と real と mpvar に定義されています．総和演算子である「`prod`」（積）,「`minimun`」（最小値）,「`maximum`」（最大値）は整数値と実数値で使えます．

```
x*5.5+(2+z)^4+cos(12.4)
  sum(i in 1..10) (min(j in s) t(i)*(a(j) mod 2))
```

string 式

文字定数は単引用符（ ’）か複引用符（ ”）で囲まなければなりません．複引用符で囲まれた string は C 言語のようなエスケープシーケンスで，バックスラッシュ文字に続きます.（￥a,￥b,￥f,￥n,￥r,￥t,￥v）

各シーケンスは対応する制御文字に置き換えられる（例えば,「￥n」は「改行」コマンドです.）か，制御文字が存在しない場合は二番目の文字自身に置き換えられます（例えば,「￥￥」は「￥」に置き換えられます.）.

エスケープシーケンスは，単引用符で囲まれている場合にはインタープリットされません.

例）

’C：￥ddd1￥ddd2￥ddd3 ’は C：￥ddd1￥ddd2￥ddd3 として理解される.

”C：￥ddd1￥ddd2￥ddd3 ”は C：ddd1ddd2ddd3 として理解される.

string には二つの基本的な演算子があります.「＋」で表される連結と「－」で表される差分です.

”a1b2c3d5 ”＋ ”e6 ”! ＝ ””a1b2c3d5e6 ”

’a1b2c3d5 ’－ ”3d5 ”! ＝ ””a1b2c ”

集合の式

定数集合は次のようなコンストラクトの一つを使って表されます.

{[Exp1 [, Exp2 …]]}

または

[{] Integer_exp1 .. Integer_exp2 [}]

最初の形式は，集合に含まれる全ての値を列挙します．整数の集合だけに鹿使えない 2 番目の形式は，整数値の間隔を与えます．この形式は，範囲集合を暗示的に定義します.

基本的な集合の演算子は和「＋」と差「－」と積「＊」です.

総和演算子「union」と「inter」も使うことができます.

```
{1,2,3}+{4,5,6}-{5..8}*{6,10} ! = {1,2,3,4,5}
{ ’a ’, ’b ’, ’c ’}*{ ’b ’, ’c ’, ’d ’} ! = { ’b ’, ’c ’}
 union(i in 1..4|i<>2) {i*3} ! = {3,9,12}
```

ブール式

ブール式の結果は，真か偽のいずれかの値をとります．伝統的な比較演算子「＜」,「＜＝」,「＝」,「＜＞」（等しくない）,「＞＝」「＞」は，整数値と実数値に対して定義されます.

これらの演算子は，string 式にも定義されています．この場合，順番は ISO-8859-1 文字集合で定義されています.（つまり，ほとんどの場合，句読点＜大文字＜小文字＜アクセント文字）

集合の場合，比較演算子「＜＝」（部分集合である）,「＞＝」（それを含む集合である）,「＝」（内容が等しい）,「＜＞」（内容が異なる）が定義されています．比較演算子は，2 つの同じ型の集合に対して使われ

なければいけません．また，演算子「expr in Set_expr」は expr が集合 Set_Expr に含まれるときに真です．逆に,「not in」演算子も定義されています．

ブール式を組み合わせるには，演算子「and」(論理和) と「or」(論理積)，単項演算子「not」（否定）が用意されています．算術式の評価は，値が既知になると同時に停止します．

総和演算子「and」と「or」は，二値の演算子に対する，これらの自然な拡張です．

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

線形制約式

線形制約条件は決定変数（mpvar 型）変数をもつ一次式を用いて作られます．

線形制約条件の形式には，以下のようなものがあります．

Linear_expr

または

Linear_expr1 Ctr_cmp Linear_expr2

または

Linear_Expr SOS_type

または

mpvar_ref mpvar_type1

または

mpvar_ref mpvar_type2 Arith_expr

最初の形式の場合には，定数は「制約されておらず」，単なる一次式です．2 番目の形式の場合は，比較演算子は「＜＝」,「＞＝」,「＝」を使うことができます．3 番目の形式は，特別な順序つきの集合を宣言するために使います．型は `is_sos1` と `is_sos2` になります．最後の 2 つの形式は，決定変数の特別な型を作るために使います．1 番目は,「`is_continuous`」,「`is_integer`」,「`is_binary`」「`is_free`」で，特別な情報は必要ありません．2 番目は，算術式によって閾値をつくるものです.「`is_partint`」は整数の一部で，整数値をとり，上限があり，連続した値をとります.「`is_semicont`」「`is_semint`」は，その変数のドメインに最低値を与えます．ひとつの変数のこれらの制約条件は，一般的な線形制約条件と同じように考えられることに注意してください．

```
3*y+sum(i in 1..10) x(i)*i >= z-t
t is_integer
sum(i in 1..10) x(i) is_sos1
y is_partint 5
```

全ての線形制約条件は，本質的に同じ形式，つまり，一次式（定数項を含む）と定数型（右項は常に 0）で格納することができます．これは，定数式「3*x>=5*y-10」は,「3*x-5*y+10」と型「同じかそれより大きい」を使ってように表すことができることを意味します．式の中で線形制約条件を参照する場合，その式の値は式が含む一次式になります．例えば，識別子「ctl」が線形制約式「3*x>=5*y-10」を参照しているとき，式「z-x+ctl」は「z-2*x-5*y+10」に等しくなります．

単項演算子の場合,「x is_type threshold」の値は「x-threshold」で,「x is_integer」の値は「x-MAX_INT」です．

タプル

タプルは中括弧で囲まれた式のリストです．

[Exp1 [, Exp2...]]

タプルは主に配列を初期化するために用います．関数，手続きのパラメータの中での配列の置き換えにも使うことができます．

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

タプルには何も演算子が定義されていません．また，タプルを値としてとるような識別子は宣言することができません．

2.8 文

Mosel 言語では，次のような 4 つタイプの文が用意されています．単純な文は，基本的な演算を行うものです．初期化ブロックは，ファイルからデータを読み込んだりファイルにデータを保存したりするために使います．選択文は，条件によって違う文の集合を選ぶことができます．繰り返し文は，演算を繰り返すのに使います．

これらの構成はひとつの文と考えられます．文のリストは文の系列です．

文と文の間には，文が式で終わっていない限り，特別な文を分割するものは必要ありません．文が式で終わっている場合には，式は改行か「；」で終わる必要があります．

単純な文

代入

代入は識別子の値を変えます．代入の一般的な形式は以下の通りです．

ident_ref := *Expression*
ident_ref + = *Expression*
ident_ref − = *Expression*

ここで，*ident_ref* は値の参照（例えば，識別子か配列の再参照），*Expression* は *ident_ref* とコンパチブルな型の式です.「直接代入」は「：＝」と書かれ，*ident_ref* の値を式の値に置き換えます.「加算代入」は「＋＝」,「減算代入」は「－＝」と書かれ，直接代入と加算，減算の組み合わせです．式の型がこれらの演算子をサポートしている必要があります.（例えば，ブーリアンのオブジェクトに加算代入を用いることはできません.）

加算代入，減算代入は線形制約式にとって，代入された識別子の制約式の型を保つという意味で，特別な意味を持ちます．通常は，式の中で使われる制約式は，それが含む一次式の値をとり，制約式の型は無視されます.

```
c:= 3*x+y >= 5
c+= y ! c が 3*x+2*y-5 >= 0 であることを意味する.
c:= 3*x+y >= 5
c:= c + y ! c が 3*x+2*y-5 であることを意味する.（c は何も制約しない）
```

直接代入はタプルを使って配列を初期化するのにも使われます.

```
T:=[2,4,6,8]
T(12):=[7,8,9,19]
```

暗示的な宣言について

各記号は使う前に宣言しなければいけません．しかし，新しい記号にあいまいでない型の値が代入される場合には，暗示的に宣言されます.

```
!A,S,SE をまだ宣言されていない記号とする.
A:=1        !A は自動的に整数として定義される.
S:={1,2,3}!S は自動的に整数の集合として定義される.
SE:={}      !これは SE の型が未知であるパースエラーになる.
```

配列の場合，暗示的な宣言は避けたほうがよいです．配列を暗示的に宣言すると，Mosel がコンテキストから添字集合を推論し，自動的に動的なサイズをもつ配列を作るので，特別な場合に用いるべきです．結果は必ずしも期待通りにはなりません.

```
A(1):=1   !「A は整数の配列 (1,..1)」 を意味する.
A(t):=2.5!「t は 1 から 10 | f(t)>0」を仮定する.
          !「A：実数の動的な配列」を意味する.
```

noimplicit オプションは暗示的な宣言を禁止します.

線形制約式

線形制約式は識別子に代入できますが，それ自身でも示すことができます．この場合，制約は「無名」といわれ，制約を既に定義している集合に付け加えられます.「名前のある制約条件」との違いは，無名の制約条件を参照すること，例えば，変更することなどができないということです.

```
10<=x; x<=20
x is_integer
```

手続き呼び出し

ソースファイルの中で全ての必要な動作がコーディングされている訳ではありません．言語には，特定の動作（メッセージを表示することなど）を行うあらかじめ定義された手続きがあります．それは，モジュールを使って，外部のロケーションから手続きをインポートすることによっても可能です（”コンパイラ命令”参照）．一般的な手続き呼び出しは以下のような形式をとります.

procedure_ident
procedure_ident (Exp1 [, Exp2 ···])

ここで，*procedure_ident* は手続きの名前で，必要な場合は *expi* が i 番目のパラメータです．あらかじめ定義された手続きの総合的なリストについては**??**章を参照してください.

`writeln("hello!")` !「hello!」というメッセージを表示する.

初期化ブロック

初期化ブロックは基本的な型のオブジェクト（スカラー，配列，集合）をテキストファイルから初期化したり，オブジェクトの値テキストファイルに保存するために使います．初期化ブロックをデータの初期化に使うときの形式は以下の通りです.

`initializations from` *Filename*
  ident1 [`as` *Label1]*
  [ident2 [`as` *Label2]...]*
`end-initializations`

ここで，*Filename* は文字列であらわされる読み込むファイルの名前で，*identi* は初期化するオブジェクトの識別子です．各識別子は自動的にラベルに関連付けられます．デフォルトでは，ラベルは識別子はそのものですが，string 式 *Labeli* で明示的に違う名前に定義することもできます．初期化ブロックが実行されたら，与えられたファイルがオープンされ，そのファイルの中で要求されたラベルが探されて，対応するオブジェクトを初期化するのに用いられます.

初期化のファイルは，以下のような形式の一つまたは複数の行を含んでいる必要があります.

Label: value

ここで，*Label* は文字列で，*value* は基本的な型（integer,real,string,boolean）の定数か，大括弧 [] で囲まれて，スペースで区切られている値の羅列です．値の羅列は，集合や配列を初期化するのに使います．ス

カラーに対して要求された場合，列の最初の値が選ばれます．配列に使われる場合には，対応する配列の場所を特定するために，丸括弧（）で囲まれた添字が値のリストに挿入されます．

以下のことにも注意しましょう．

- 特別な書式は必要ありません．スペース，タブ，行の中断がセパレータです．
- 一行のコメントがサポートされています．(すなわち,「！」で始まり，行の終わりで終了します．
- ブーリアンコンストラクトは，識別子「false」と「true」，または，定数「0」「1」です．
- 全ての文字列（ラベルを含む）は，単引用符または複引用符で囲まれます．複引用符の場合，エスケープシーケンス（すなわち,「¥」を使用）がインタープリッとされます．

初期化ブロックの 2 番目の形式は，データをファイルに保存するために使われます．

```
initialization to  Filename
  ident1[as Label1]
  [ident2[as Label2]...]
end-initialization
```

2 番目の書式が実行された場合，全てのラベルの値は，与えられたファイルの中の対応する識別子の現在の値にアップデートされます[1]．ラベルが見つからない場合には，新しくファイルの終わりに記録され，ファイルがまだ存在していない場合には，新しくファイルが作られます．

例えば,「a.dat」が次のような内容であるとします．

```
! Example of the use of initialization blocks
t:[ (1 un) 10 (2 deux) 20 (3 trois) 30 ]
t2:[ 10 (4) 30 40 ]
 'nb used ': 0
```

次のようなプログラムを考えます．

```
model "Example initblk"
  declarations
    nb_used:integer
    s: set of string
    t: array(1..3,s) of real
    t2: array(1..5) of integer
  end-declarations
  initializations from  'a.dat '
    t ! t=[(1,' un ',10),(2,' deux ',20),(3,' trois ',30)]
    t2 ! t2=[10,0,0,30,40]
```

[1]アップデートする前に，オリジナルのファイルのコピーが保存されます．(すなわち，オリジナルが「fname であったら,「fname-」に保存されています．)

```
    nb_used as "nb used" ! nb_used=0
  end-initializations
  nb_used+=1
  t(2,"quatre"):=1000

  initializations to  'a.dat '
    t
    nb_used as "nb used"
    s
  end-initializations
end-model
```

このプログラムの実行後に，データファイルは以下のようになります.

```
! Example of the use of initialization blocks
t:[(1  'un ') 10 (2  'deux ') 20 (2  'quatre ') 1000 (3  'trois ') 30]
t2:[ 10 (4) 30 40 ]
 'nb used ': 1
 's ': [ 'un '  'deux '  'trois '  'quatre ']
```

条件分岐文

If 文

一般的な「if」文は以下のような形式です.

```
  if Bool_exp_1
  then Statement_list_1
  [
    elif Bool_exp_2      then Statement_list_2   ]
  [else Statement_list_E
]   end_if
```

条件分岐文は以下のようにして実行されます．もし *Bool_exp_1* が「true」なら，*Statement_list_1* が実行され，end-if 命令の後に続きます．そうでない場合，elif 文があれば，if の場合と同様に実行されます．もし，全てのブール式が「false」で else 命令があれば，*Statement_list_E* が実行され，end-if 命令の後に続きます.

```
if c=1
then writeln( 'c=1 ')
elif c=2
then writeln( 'c=2 ')
else writeln( 'c<>1 and c<>2 ')
```

```
end-if
```

Case 文

一般的な「Case」文は以下のような形式です.

```
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
```

条件分岐文は以下のようにして実行されます. まず *Expression_0* を評価し, 順番に一致するまで *Expression_i* と比較します. 一致した式に対応する *Statement_i*（*Statement_list_i* 文のそれぞれの文）を実行し, end_case 命令の後に続きます. 一致するものがなく, else 命令があれば, *Statement_list_E* が実行され, そうでなければ end-if 命令の後に続きます. 式のリスト *Expression_i* のそれぞれは, スカラー, または, 集合, または, カンマで区切られた式の集合であることに注意してください. 最後の 2 つの場合, *Expression_0* 式が集合の要素に対応するか, リストの要素に対応すると, 一致したとみなされます.

```
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
```

ループ

forall ループ

一般的な「forall」文は以下のような形式です.

forall(*Iterator_list*) *Statement*

または,

forall(*Iterator_list*) do *Statement_list* end-do

Statement 文（*Statement_list* 文のそれぞれの文）を繰り返しのリストによって作られた添字のタプルそれぞれについて繰り返します (「総和演算子」参照).

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

While ループ

一般的な「`While`」文は以下のような形式です.

`while`(*Bool_expr*) *Statement*

または,

`while`(*Bool_expr*) `do` *Statement_list* `end-do`

Statement 文（*Statement_list* 文のそれぞれの文）を, *Bool_expr* が「`true`」である間，繰り返します．最初に評価したときに「`false`」なら，while 文全体がスキップされます.

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

Repeat ループ

一般的な「`repeat`」文は以下のような形式です.

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

`repeat` と `until` で囲まれている文のリストを，*Bool_expr* が「`true`」である間，繰り返します.「`while`」ループとは逆に，文は少なくとも一回は実行されます.

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

break 文と next 文

break 文と next 文は，それぞれ，ループを中断し，次の繰り返しのループに飛ぶために使われます．一般的な「break」文「next」文は以下のような形式です．

break [n]

または，
next [n]

ここで，n は，オプションの整数定数で，*n-1* 番目にネストされたループを演算をする前に停止します．

```
! ループ制御の例
repeat                !1:ループ L1
  forall (i in S) do !2:ループ L2
    while (C3) do     !3:ループ L3
      break 3         !4:L3 をやめて 11 行目のあとから続ける
      next            !5:L3 の次の繰り返しへ (3 行目)
      next 2          !6:L3 をやめて次の 'i 'へ (2 行目)
    end-do            !7:L3 の終わり
    next 2            !8:L2 をやめて L1 の次の繰り返しへ (11 行目)
    break             !9:L2 をやめて 10 行目のあとから続ける
  end-do              !10:L2 の終わり
until C1              !11:L1 の終わり
```

2.9 手続きと関数

サブルーチンの形で文と宣言の集合を集めることが可能で，いったん定義されると，モデルの実行の間に何度も呼び出すことができます．Mosel には 2 種類のサブルーチン，手続きと関数があります．手続きは文の中で用いられ (例えば「writeln("Hi!")」)，関数は文の一部です (なぜなら，値が返されます．例えば,「round(12.3)」)．手続きも関数も，引数をとり，ローカルなデータを定義し，再帰的に呼び出すこともできます．

定義

サブルーチンを定義することは，その外部のプロパティ (すなわち，その名前と引数)，動作を示すことです．そのサブルーチンが実行される (すなわち，実行すべき文) ときに定義されます．一般的な手続きの定義の形式は以下の通りです．

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

ここで，*name_proc* は手続きの名前で *list_of_args* は（もしあれば）引数です．このリストはカンマで区切られた記号の宣言です（2.6 「宣言ブロック」参照）．通常の宣言との違いは，配列の添字リストに定数や式が含まれてはいけないことです (例えば,「A-12」や「t1:array(1..4) of real 」などは，引数の宣言では適切ではありません)．手続きの本体は普通の文のリストで，宣言ブロックには包含できる手続きや関数の定義を書いてはいけません．

```
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 ! myproc の呼び出し
withparams(mytab,23,67) ! withparams の呼び出し
```

関数の定義は手続きの定義とよく似ています．

```
function name_func [(list_of_args)]: Basic_type
  Func_body
end-function
```

手続きとの違いは，関数の型を決めなければいけないところだけです．Moselは，基本的な型（integer,real,boolean,string）の関数しかサポートしません．関数の本体の中では，関数の型の特別な変数「returned」が自動的に定義されます．この変数は，関数の返り値として用いられます．それゆえ，関数の実行中に値を代入する必要があります．

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

writeln("3*12=",multiply_by_3(12)) ! 関数の呼び出し
```

引数：値の引渡しの規則

基本的な型の引数は値で渡され，他の全ての型の引数は参照によって渡されます．引数が値によって渡される場合は，そのサブルーチンがこの引数の値を変えたとしても，呼び出した元の所では引数の値は変わ

りません．そのサブルーチンは，情報のコピーを受け取ります．しかし，引数が参照によって渡される場合には，そのサブルーチンは引数そのものを受け取ります．その結果，その引数の値がサブルーチンのプロセスの間に変えられたら，呼び出した元の所でも同じく値が変わります．

```
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
```

ローカルな宣言

いくつかの宣言ブロックは，サブルーチンに使われる可能性があり，宣言された全ての識別子はこのサブルーチンにローカルなものです．これは，これらの記号の全てがサブルーチンの有効範囲 (つまり，宣言したところから `end-procedure` または `end-function` までの間) にのみ存在することを意味します．そして，それらが問題の一部でない限り，サブルーチンの実行が終了すれば，それらのリソース全てが開放されます． それゆえ，決定変数 (`mpvar`) とアクティブな線形制約式 (ただの線形制約式ではない `linctr`) は，保存されます．結果として，サブルーチンの中で宣言されたあらゆる決定変数，または，制約式は，たとえ関連のオブジェクトを指定するために使われる記号がそれ以上定義されないとしても，サブルーチンの終了後にも有効です．

ローカルな宣言がグローバルな記号を隠すかもしれないことに注意してください．

```
declarations ! グローバルな定義
  i,j:integer
end-declarations

procedure myproc
  declarations
    i:string ! グローバルな記号を隠す宣言
  end-declarations
  i:="a string" ! ローカルな 'i'
  j:=4
  writeln("Inside of myproc, i=",i," j=",j)
end-procedure

i:=45 ! グローバルな 'i'
j:=10
```

```
myproc
writeln("Outside of myproc, i=",i," j=",j)
! displays:
! myproc の中では，i=a string j=4
! myproc の外では，i=45 j=4
```

オーバーロード

Mosel は手続きと関数のオーバーロードをサポートします．同じ関数を引数の異なるセットで複数回定義することができます．コンパイラは，引数リストに応じてどちらのサブルーチンを使うかを決定します．あらかじめ定義された手続きと関数にも適用されます．

```
! 1 から与えられた上限までの間の乱数を返す
function random(limit:integer):integer
  returned:=round(.5+random*limit) !あらかじめ定義された random 関数を使う．
end-function
```

以下のことは重要ですので注意しましょう．

- 手続きは関数にオーバーロードできません．逆も同じです．
- どのような識別子も再定義することはできません．この規則は，手続きと関数にも適用されます．サブルーチンの定義は，少なくとも 1 つの引数が異なりさえすれば，別のサブルーチンにオーバーロードするために使うことができます．関数の返り値の型が違うだけでは，オーバーロードできません．

フォワード宣言

ソースファイルをコンパイルしている間，それより前に宣言された記号だけを，あらゆる場所で利用することができます．2 つの手続きが再帰的に自身を呼び出す場合には (クロス回帰)，前もって 2 つの手続きのうちの 1 つを宣言する必要があります．更に，より明瞭なものとするためには，ソースファイルの終りに全ての手続きと関数の定義を集めることは，しばしば有益です．フォワード宣言は，これらのために提供されます．それは，後で定義されるサブルーチンのヘッダのみを書きます．一般的なフォワード宣言の形式は以下の通りです．

`forwarf procedure` *Proc_name[(list_of_prams)]*
`forwarf procedure` *Func_name[(list_of_prams)]* :*Basic_type*

ここで，手続き *Proc_name* または関数 *Func_name* はソースファイルの後のほうで定義されます．フォワード宣言された関数の定義がない場合には，Mosel にエラーであると判断されることに注意してください．

```
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
```

2.10 public 装飾子

いったんソースファイルがコンパイルされると，Mosel では，モデルのオブジェクトを示すために用いる識別子は使えない状態になります．モデルが実行された後で (たとえばコマンドラインインタープリタの DISPLAY コマンドを使って)，情報にアクセスするために，記号の表が BIM ファイルに保存されます．ストリップオプション「-s」をつけてソースファイルのコンパイルを行った場合，全てのプライベートな記号は記号の表から除かれます．デフォルトでは，すべての記号がプライベートとみなされます．

public 修飾子は，オブジェクトの宣言と定義に使われます．修飾された識別子 (パラメータやサブルーチンを含む) は，ストリップオプションがある場合にも記号の表に記録されます．

```
parameters
  public T="default" ! T は記録
end-parameters

declarations
  public a,b,c:integer ! a,b,c は記録
  d:real ! d はプライベート
end-declarations

forward public procedure myproc(i:integer)  ! ‘ myproc ’は記録
```

2.11 入出力の扱い

プログラム/モデルの実行の最初に，標準入力ストリームと標準出力ストリームの 2 つのテキストストリームが自動的に作成されます．標準出力ストリームは，テキストを書く手続き (write，writeln,fflush) によって使われます．標準入力ストリームは，テキストを読む手続き (read，readln，fskipline) によって使われます．これらのストリームは，Mosel が実行されている環境から継承されます．通常，出力手続きは，何かをコンソールにプリントすることを意味し，入力手続きは，ユーザによってキーボードから何かをタイプされるのを待つことを意味します．

手続き fopen と fclose は，テキストファイルを入出力のストリームにすることを可能にします．この場合，出入力関数をファイルの読み書きに使います．ファイルが開かれると，それが自動的にアクティブな (その初めの状態に従って) 入力，または，出力ストリームになりますが，ストリームに以前に割り当てられたファイルが開いたままの状態になることに注意してください．手続き fselect と関数 getfid の組み合わせで，違うファイルを開いて，スイッチすることができます．

```
model "test IO"
def_out:=getfid(F_OUTPUT) ! ファイル ID をデフォルト出力に保存する．
```

```
  fopen("mylog.txt",F_OUTPUT)!出力を‘ mylog.txt ’にスイッチ.
  my_out:=getfid(F_OUTPUT) ! ID を現在の出力に保存する.
  repeat
    fselect(def_out) ! デフォルト出力を選ぶ
    write("Text? ") ! メッセージを書く
    text:= ''
    readln(text) ! デフォルト入力から string を読む
    fselect(my_out) !  'mylog.txt 'を選ぶ
    writeln(text) ! string をファイルに書き込む
  until text= ''
  fclose(F_OUTPUT) ! 現在の出力 (‘ mylog.txt ’) をクローズ
  writeln("Finished!") ! デフォルトのアウトプットにメッセージを表示
end-model
```

2.12 モジュールを使った言語の拡張

Mosel 言語は多数の関数と手続きを提供すると同時に，その機能の多くはモジュール (追加の特徴を提供することによって言語を拡張する) としてのみ利用可能です．これは，更に多くの柔軟性を与えます．例えば，必要な場合には，モジュールとしてこれらの特徴をロードする特別なソルバーエンジン，あるいは，データベース接続の技術と Mosel を切り離すこともできます．その結果，言語が拡張可能 (ダッシュ・オプティマイゼイションによって書かれたモジュールを用いてばかりではなくユーザーの自分自身のコマンドによって増やすことができます) となります．

現在，mmive，mmodbc，mmsystem，mmxprs のモジュールが標準の Mosel ディストリビューションで提供されています．これらのモジュールで提供されているコマンドを使うには，対応するモジュールの名前をモデルファイルの最初に次のように書きます．

```
uses "module_name"
```

これらのいくつかについて追加の制御と属性が提供されます．詳細が第 6，7 章のリファレンスに書いてあります．この節の残りで，各々のこれらのモジュールについて簡潔に示します．

mmsystem

mmsystem モジュールは，OS に関係した手続きと関数を提供します．このようなコマンドの性質のため，動作はシステムの間で異なるかもしれないので，使用するときには注意しなければなりません．mmsystem の手続きと関数は，一般に間違いなく getsysstat を使って読むことのできる状態変数をセットします．動作が正しく行われたことを確認するために，変数の値を各システムコールの後でチェックしたほうがよいでしょう．mmsystem モジュールに含まれるコマンドは以下の通りです．

コマンド	説明	ページ
fdelete	ファイルを消去する.	
fmove	ファイルを移動する.	
getcwd	現在動作中のディレクトリを返す.	
getenv	環境変数の値を返す.	
getfstat	ファイルまたはディレクトリの状態を返す.	
getsysstat	システムの状態を返す.	
gettime	秒単位で計った時間を返す.	
makedir	ディレクトリを作る.	
removedir	ディレクトリを削除する.	
system	OS のコマンドを実行する.	

mmxprs

mmxprs モジュールは， Mosel モデルの中から Xpress-Optimizer へのアクセスを行います．従って，システムに Xpress-Optimizer モジュールがインストールされている必要があります．多数の最適化関連の命令は，問題の解を見つけることから，カット管理ルーチンのためのものまで，多岐にわたります．これらの使用方法は，言語リファレンスの第 4 章「Mosel 関数と手続き」にありますが，詳細はオプティマイザリファレンスマニュアルを見てください.

このモジュールを使うと，オプティマイザの制御と属性 (例えば XPRS_LPSTATUS) へアクセスする関数 getparam と手続き setparam が使えるようになります．次のような制御と属性が定義されます.

- XPRS_VERBOSE:オプティマイザがメッセージを表示するのを有効/無効にする.
- XPRS_LOADNAMES：オプティマイザに MPS 名をロードするのを有効/無効にする.
- XPRS_PROBLEM：オプティマイザ問題のポインタ

カットマネージャの利用: Mosel からカットマネージャを実行するためには，あるオプティマイザの制御をセット（リセット）する必要があることに注意しなければなりません.

```
setparam("XPRS_PRESOLVE",0);
setparam("XPRS_CUTSTRATEGY",0);
setparam("XPRS_EXTRAROWS",5000);
```

カットマネージャに適切なコールバック関数や手続きは，他のオプティマイザコールバックと同様に，関数 setcallback を使って初期化する必要があります.

カットは，Mosel では保存されず，ただちにオプティマイザに送られることに注意しなければなりません．従って，問題がオプティマイザに再びロードされた場合には，以前に定義されたカットは全てなくなります．Mosel では，カットは一次式 (すなわち，とりうる値の制限のない制約式)，オペレータサイン (不均等/均等) を指定することによって定義されます．一次式の代りに制約が与えられると，追加の制約としてシステムに付け加えられます.

mmxprs モジュールコマンド: mmxprs モジュールで定義されている関数と手続きは以下の通りです.

コマンド	説明	ページ
addcut	オプティマイザにおいて問題にカットを加える.	
addcuts	オプティマイザにおいて問題にカットの集合を加える.	
clearmipdir	定義された全ての MIP 命令を消去する.	
clearmodcut	定義された全てのモデルを消去する.	
delbasis	以前に保存された基底を消去する.	
delcuts	オプティマイザにおいて問題からカットを消す.	
dropcuts	カットプールからカットの集合を削除する.	
getcnlist	現在のノードでアクティブなカットの集合を返す.	
getcplist	カットプールのカットの集合を返す.	
getlb	変数の最小値を返す.	
getprobstat	オプティマイザ問題の状態を返す.	
getub	変数の上限値を返す.	
loadbasis	保存されている基底をロードする.	
loadcuts	カットプールからオプティマイザにカットをロードする.	
loadprob	オプティマイザに問題をロードする.	
maximize	目的関数を最大化する.	
minimize	目的関数を最小化する.	
savebasis	基底を保存する.	
setcallback	コールバック関数を設定する.	
setlb	変数の最小値をセットする.	
setmipdir	命令を設定する.	
setmodcut	モデルのカットとして制約式をマークする.	
setub	変数の最大値をセットする.	
storecut	カットプールにカットを格納する.	
storecuts	カットプールに複数のカットを格納する.	

mmodbc

Mosel ODBC インタフェースは，ODBC ドライバが利用可能であるスプレッドシートやデータベースにアクセスするために使うことができる手続きと関数を提供します．ODBC が今日の利用可能な最もポピュラーなデータベース/スプレッドシートプログラムの大多数によってサポートされるので，このモジュールを使うことによって，自然に保存したデータ，解法結果にアクセスすることができるようになります．以下の例をみると，このような利点がよくわかります．

データベースの例：データソースが mydata で，以下のような pricelist というテーブルである場合を考えます.

articlenum	colour	price
1001	blue	10.49
1002	red	10.49
1003	black	5.99
1004	blue	3.99

以下のような例で，データベースと Model モデルファイルを接続したり，接続を切ったりします.

```
model ‘ ODBC Example ’
  uses "mmodbc"

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

  setparam("SQLVERBOSE",true)
  SQLconnect("DSN=mydata")
  writeln("Connection number: ",getparam("SQLCONNECTION"))
  SQLexecute("select articlenum,price from pricelist",prices)
  SQLdisconnect
end-model
```

ここで，SQLVERBOSE 制御は，真である場合，エラーがあったときに ODBC メッセージを表示します. データベースと接続すると，コマンド SQLexecute が呼び出されて price フィールド（添字は articlenum）からエントリが検索され，テーブル pricelist に格納されます．最後に，接続が切られます.

mmodbc モジュールコマンド: mmdobc モジュールで定義されている関数と手続きは以下の通りです.

コマンド	説明	ページ
SQLconnect	ODBC が可能なアプリケーションに接続する.	
SQLdisconnect	ODBC が可能なアプリケーションに接続を切る.	
SQLexecute	アプリケーションで SQL コマンドを実行する.	
SQLreadinteger	アプリケーションから integer の値を読む.	
SQLreadreal	アプリケーションから real の値を読む.	
SQLreadstring	アプリケーションから string の値を読む.	

mmive

mmive モジュールは，Xpress-MPIntegrated Visual Environment （IVE）によって，グラフィック能力を拡張するために使われます．このモジュールは，現在は 2 つのコマンドをサポートしています．これらのコマンドは，ユーザーがあらゆる計算された関数のグラフを表示するためのものです.

サポートされているコマンドは以下の通りです.

コマンド	説明	ページ
IVEaddtograph	ユーザのグラフに点を加える.	
IVEinitgraph	ユーザのグラフを初期化し，軸に名前をつける.	

第 3 章

Moselの実行

3.1 コンソール Mosel

スタンドアロンバージョンの Mosel は，バッチモードまたはコマンドラインインタープリタによって一般的コマンドを実行するための，簡単なインタフェースを提供します．ユーザーは，ソースモデル，または，プログラム (「.mos」ファイル) をコンパイルしたり，バイナリモデル (「.bim」ファイル) をロードして実行したり，記号の値，行列を表示したり保存したりします．いくつかのバイナリモデルは，一度にロードされた後，順番に使うことができます．

Mosel の実行形式は次のようなコマンドラインオプションを受け付けます．

-h	短いヘルプメッセージを表示して終わります．
-v	バージョンを表示して終わります．
-s	サイレントモードです．(バッチモードのときだけ有効です．)
-c *commands*	Mosel をバッチモードで実行します．パラメータ *Commands* は，セミコロンで区切られたコマンドのリストです (これは，OS や使っているシェルによって単引用符または複引用符で囲まれるかもしれません)．*Commands* はリストの最後まで順番に実行され，Mosel は終了します．例えば，mosel -c "CLOAD -sg mymodel;RUN のようにします．

コマンドラインオプションが指定されないときは，Mosel は対話型モードでスタートします．次のコマンドはコマンドプロンプトで実行されます (大括弧 [] に納められている引数は任意です)．コマンドラインインタープリタは大文字小文字を区別しますが，より明瞭にするために大文字でコマンドを表示します．

INFO ***[symbol]***: このコマンドは，引数なしで実行された (これが問題報告にとって有益であるかもしれない) プログラムに関する情報を表示します．引数は，現在のモデルからの記号と解釈されます．要求された記号が実際に存在する場合，このコマンドは，型と構造に関する情報を表示します．

SYSTEM ***[command]***: OS のコマンドを実行します．

QUIT: 現在の Mosel セッションを終了します．

COMPILE[-sgep] ***filename[comment]***: モデル「filename」をコンパイルし，コンパイルが成功するとバイナリモデル (BIM) ファイルを生成します．ファイルに拡張子がない場合は拡張子「.mos」を足し，バイナリファイルの名前の拡張子は「.bim」となります．フラグ「-e」が選ばれた場合は，ソー

スファイルの名前に自動的に拡張子をつけないようにします．フラグ「-s」がつけられた場合は，オブジェクトの名前（例えば，変数や定数）はプライベートで，BIM ファイルに保存されません．フラグ「-g」がつけられた場合は，デバッグ情報をつけます．これは，ランタイムエラーの場所を調べるのに必要です.「*comment*」オプションパラメータは BIM ファイルにコメントをつけるのに使います.（「LIST」コマンド参照）フラグ「-p」が選ばれた場合は，ソースファイルの文法チェックだけが行われ，コンパイルはされず，ファイルは出力されません．

LOAD ***filename***: 「filename」という名前の BIM ファイルをロードし，それを実行するために必要な全てのモジュールを開きます．拡張子がなければ,「.bim」という拡張子をつけます．同じ名前のモデルがすでに現在のコアメモリにロードされている場合には，新しいものに置き換えられます（モデルの名前は model 文によって決定されます．ファイル名である必要はありません.）.

CLOAD[-sge] ***filename[comment]***: 「*filename*」という名前のファイルをコンパイルし,（コンパイルが成功すれば）結果のファイルをロードします．このコマンドは「COMPILE *filename*」と「LOAD *filename*」をこの順番で実行するのと等価です．

LIST: CLOAD または LOAD を使ってロードされた全てのモデルを表示します．それぞれのモデルについて，以下の情報が表示されます．

- name: モデルの名前（ソースファイルの中で model 文によって与えられたものです.）です．
- number:モデル番号はモデルがロードされるときに自動的に割り当てられます．
- size:モデルによって使用しされたメモリの量（バイト数）です．
- system:ソースファイル名を示すコンパイラによって生成された文字列で，モデルがデバッグ情報，かつ/または，記号を含むかどうかをあらわします．
- user comment:コンパイル時にユーザーによって定義されたコメントです (COMPILE,CLOAD 参照).

アクティブなモデルは，名前の前にアスタリスク（「＊」）がつけられます（DELETE,RUN.RESET コマンドによってアクティブになります.）．デフォルトでは，最後にロードされたモデルがアクティブです．

SELECT ***number—name***: モデルをアクティブにします．モデルは，名前か命令番号のいずれかを用いて選択することができます．モデルのリファレンスがない場合には，現在アクティブなモデルに関する情報が表示されます．

DELETE ***number—name***: メモリからモデルを削除します (BIM ファイルはこのコマンドによって影響を受けません)．モデルの名前かシーケンス番号が与えられないときは，アクティブなモデルが削除されます．アクティブなモデルが削除されると，一番最後にロードされたモデル (もしあれば) が新しくアクティブなモデルになります．

RUN ***[parameters]***: アクティブなモデルを実行します．オプションで，モデルと/またはモジュールの制御パラメータを初期化するためにパラメータの値を与えることができます．初期化の文法は．モデルパラメータは「$param\_name = value$」のように設定し，コントロールパラメータの場合は，*dsoname* をモジュール

の名前，*ctrpar_name* をセットするコントロールパラメータの値として,「*dsoname.ctrpar_name=value*」のように行います.

RESET: 全てのリソースを解放してアクティブなモデルを再び初期化します.

EXPORTPROB[-[ms] [***filename[objective]*** :] アクティブな問題に対応する行列を表示，または,（*filename* で指定されたファイルに）保存します．行列は，LP 形式か MPS 形式 (-m オプション) で出力します．モデルの実行後にも問題が入手可能です．フラグは最適化のやり方（「-p：最大にする）とファイル形式（「-m」: MPS 形式にする）の選択に，オブジェクトの名前を使うかどうか（「-s」: スクランブルされた名前–これは，オブジェクトの名前が得られていないときにはデフォルトです.）*objective* は定数の名前を決めるのに使うことができます.

DISPLAY ***symbol***: 与えられた記号の値を表示します．モデルを実行する前には定数だけにアクセスすることができます．決定変数の場合には，解の値 (デフォルトは 0)，制約式の場合は，アクティビティの値 (デフォルトは 0) が表示されます.

SYMBOLS [-cspo]: 現在のモデルによって公表されている記号のリストを表示します．フラグはどのような種類の記号を表示するかのフィルタとして利用することもできます.「-c」は定数,「-s」は手続き/関数,「-p」は制御/属性,「-o」は他の全てです.

LSLIBS: ロードされている全ての動的共有オブジェクト (DSO) と各モジュールを，バージョン番号と参照数 (すなわち，それを使うモジュールの数）を表示します.

EXAMINE [-cspt] ***libname***: モジュール *libname* の定数，手続き/関数，型，制御/属性のリストを表示します．オプションのフラグはどの情報を表示するかを選ぶのに使います.「-c」は定数,「-s」は手続き/関数,「-t」は型,「-p」は制御/属性です.

FLUSHLIBS: 全ての使っていない動的共有オブジェクトをアンロードします.

コマンドが理解されなかった場合，可能なキーワードとその短い説明のリストが表示されます．コマンドの名前は，不確かさがない限り短くすることができます（たとえば,「CL」を「CLOAD」の代わりに使うことができますが,「C」は COMPILE と区別できないので十分ではありません.）．文字列の引数は，単引用符または複引用符で囲まれます[1]．文字列が数字で始まる場合やスペースと/または引用符で始まる場合には，囲む必要があります.

典型的には，モデルは次のようなコマンドでロードされて実行されます.

```
CLOAD mymodel
RUN
```

使うためのより進んだ情報は，Xpress-MP エッセンシャルガイドを参照してください.

[1] パラメータ 10 は数ですが，”10 ”や ’10 ’は文字列です.

3.2 Mosel のライブラリ

Mosel ライブラリは，C のようなプログラミング言語において開発されたアプリケーションに Mosel 環境を埋め込むために使うこともできます.

以下のようなことができるようになる機能が提供されています.

- ソースモデルファイルをコンパイルしてバイナリモデル (BIM) ファイルにします.
- 一度にいくつかのモデルを扱う BIM ファイルをロードしたり，アンロードしたりします.
- モデルを実行します.
- 処理後のインタフェースを通じて，Mosel の内部のデータベースにアクセスします.
- Mosel によって使われる動的な共有オブジェクトを管理します.

2 つのライブラリが提供されています．第 1 は，ランタイムライブラリで，既にコンパイルされているモデルをロードして実行するのに必要な機能を含んでいます．第 2 は，モデルコンパイラライブラリで，ソースモデルファイルからバイナリモデルファイルを作るために使われる Mosel コンパイラです．一般に，最初のライブラリがアプリケーションバイナリフォーム (Mosel によって実行可能な形式) で提供されるモデルに使われます.

この文章は，2 つのライブラリに含まれる全ての関数を記述します．Mosel ライブラリとプログラムをコンパイルしてリンクする方法に関する更に多くの詳細は，このソフトウェアのディストリビューションの例を参照するか，Xpress-MP エッセンシャルガイドを参照してください.

ランタイムライブラリ Mosel ランタイムライブラリ（`xprm_rt`）は，BIM 形式のモデルをロードしたり実行したり，モデルオブジェクトにアクセスするために必要な関数の集合を提供します.

このライブラリを使うプログラムは，ヘッダファイル `xprm_rt.h` を包含する必要があります．このヘッダファイルは，以下のような型を定義します.

`XRPMMmodel`: コアメモリにストアされているモデルへのリファレンス.

`XPRMdsolib`: 動的共有ライブラリのディスクリプタへのリファレンス.

`XPRMmpvar`: 決定変数へのリファレンス.

`XPRMlinctr`: 線形制約式へのリファレンス.

`XPRMset`: 集合へのリファレンス.

`XPRMarray`: 配列へのリファレンス.

`XPRMproc`: 手続き/関数へのリファレンス.

次のような基本的な型も定義されています.

`XPRMinteger`: 整数値 (C の `int` 型).

`XPRMreal`: 実数値 (C の `double` 型).

`XPRMboolean`: ブール値 (C の `int: 0 = false, 1 = true`)

`XPRMstring`: 文字列値 (C の `const char *`型)

モデルコンパイラライブラリ Mosel モデルコンパイラライブラリ (xprm_mc) は，Mosel のコンパイラから成ります．提供される機能は，ソースプログラムを対応するバイナリモデル（BIM）ファイルにコンパイルするときにのみ使われます．xprm_mc は，XPRMcompmod が呼ばれる前に，ランタイムライブラリ xprm_rt が存在し，初期化されている必要があることに注意してください．

モデルコンパイラライブラリを使うためには，ヘッダファイル xprm_mc.h を包含する必要があります．

3.3 Mosel ライブラリのルーチン

初期化と終了 Mosel ライブラリを使ったプログラムは，最初に初期化ルーチン XPRMinit を呼ばなければなりません．Mosel ライブラリを実行時に動的にロードしたりアンロードしたりする場合には，Mosel の使っているリソースを解放するためのライブラリをアンロードし，終了ルーチン XPRMfree を呼び出します．

モデルの管理 コアメモリにロードされたモデルを操作するために，たくさんの関数が提供されています．ロード (XPRMloadmod)，実行 (XPRMrunmod)，アンロード (XPRMunloadmod)，情報を得る (XPRMgetmodinfo) ための関数，などです．Mosel のひとつのセッションで複数のモデルをロードし，交互に使うこともできます．ロードされているモデルのうちどのモデルを使うかを設計するために，各関数には問題ポインタ (MPRMmodel 型) の引数が必要です．関数 XPRMloadmod は，BIM(バイナリモデル) ファイルからモデルを読み込むのに成功すると，XPRMmodel 型のオブジェクトを返します[2]．

処理後のインタフェース 処理後 (pp) のインタフェースによって，Mosel 内部のデータベースにアクセスするのが容易になります．このデータベースは，BIM ファイルで（定数のように）定義され，(配列のように）モデルの実行中に作られた全てのモデルオブジェクトから成ります．動的に作られるオブジェクトは，モデルが実行された後でなければ利用できません．データベースから，(関数 XPRMfindident を使って）エントリを探したり，(関数 XPRMgetnextparam を使って）パラメータを得たり，(関数 XPRMgetnextproc を使って) 手続きのバージョンを得たりすることができます．

モデルがオプション「s」（ストリップ記号）をつけてコンパイルされ，明示的に記録された記号がなかった場合（宣言の public 修飾子の説明参照）には，データベースは利用できません．そのようなモデルには，処理以後のインタフェースを通じてアクセスすることができません．

集合 集合は配列の添字のために使います．配列を使うモデルは，明示的に集合が定義されていなくても，集合を使います．範囲は整数の集合の特殊な場合で，与えられた範囲の全ての連続した整数を含むことに注意してください．

以下のような情報にアクセスするための手続きと関数が提供されています．集合の型 (XPRMgetsettype)，サイズ (XPRMgetsetsize)，
要素の値と添字 (XPRMgetelsetval，XPRMgetelsetndx，XPRMfirstsetndx,XPRMlastsetndx) です．

配列 Mosel では，配列は，他の配列や集合を含むあらゆる種類のオブジェクトに使うことができます．集合の場合と同様に，配列のサイズ，(XPRMgetarrsize)，次元 (XPRMgetarrdim)，エントリ (XPRMfirstarrentry，XPRMnextarrentry,XPRMlastarrentry)，その値 (XPRMgetarrval) を決定するためのルーチンがあります．

[2]BIM ファイルは Mosel コンパイラによって，コマンドインタープリタまたはモデルコンパイルライブラリを使って生成されます．

配列の型は，集められているオブジェクトの型になるか，XPRMgetarrtype ルーチンで決定することもできます．Mosel が現在サポートしている型は，XPRM_TYP_INT，XPRM_TYP_REAL，XPRM_TYP_STRING，XPRM_TYP_BOOL，XPRM_TYP_MPVAR，XPRM_TYP_LINCTR，XPRM_TYP_SET，XPRM_TYP_ARR です．記憶クラスはどのようにメモリにオブジェクトがストアされているかを表します．多くの場合，この情報は自動的に特殊な場合を扱うことのできる配列にアクセスする関数のように，無視することも可能です．

記憶クラスは 2 ビットにエンコードされます．

XPRM_GRP_DYN: 配列は動的です．添字集合には範囲が定義されていません（すなわち，この配列では,「out of range error」のエラーは起こらず，添字集合はオンデマンドで大きくなります.）.

XPRM_GRP_GEN: 配列は一般的です (すなわち，動的に制限されています．)．要素数は作られるときに決められた範囲のリミットまで大きくなります．

「疎なテーブル」には，記憶クラス XPRM_GRP_DYN か，XPRM_GRP_DYN|XPRM_GRP_GEN (動的または固定の範囲) を使います．Mosel コンパイラはそれぞれの配列がどの記憶クラスを使うべきかを決定します.「密なテーブル」でも，モデルがコンパイル時に実際の配列のサイズを判断できない場合には，記憶クラス XPRM_GRP_DYN が使われることもあります．

動的な配列では，ロジカルなエンティティと真のエンティティとは区別されます．範囲 1 .. 5 で作られ，エントリ 3 だけが定義された配列を考えると，5 個のロジカルなエントリがありますが，実際にはひとつのエントリしかありません．この違いは，列挙されている配列に提供た関数において主に注意する必要があります．Mosel には真の配列の要素にアクセスするための 2 つの特別な関数（XPRMfirstarrtruentry,XPRMnextarrtruentry）が用意されています．

ライブラリレベルでは，全ての配列は整数で添字化されています（負の値も許します）．テキストの添字の値を使うためには，テキストから数字に変換するための関数 XPRMgetelsetndx を使う必要があります．

問題　モデルは，アクティブな問題のコンテクストにおいて実行されます．デフォルトでは，モデルを処理する一番最初に，初期の問題「main problem」が作られます．Mosel は,「main problem」のコンテクストでで作られたり使われたりした，線形制約式や決定変数に関係する様々な情報にアクセスするための関数の集合を提供します．特に，目的関数の値（XPRMgetobjval），変数の解の値 (XPRMgetvsol)，制約式の左辺の値 (XPRMgetcsol)，減らされたコスト (XPRMgetrcost)，双対値 (XPRMgetdual)，スラック値 (XPRMgetslack) が返されます．

関数 XPRMexportprob を除き，この節の全てのオペレーションは，明示的に（たとえば，mmxprs モジュールの手続き loadprob），または，モデルの最適化オペレーションを用いて暗示的（例えば，mmxprs モジュールの手続き maximize）に，問題をソルバーにロードする必要があります．何の問題もない場合（モデルが実行されていない，制約条件がモデルによって作られていない，または問題がソルバーにロードされていない），各関数はデフォルトの値を返します．

モジュールの取り扱い　Mosel の機能は，動的な共有オブジェクト（DSOs）として実装されているモジュールを使って拡張することができます（2.12 節「モジュールを使った言語の拡張」参照）．Mosel のモジュールマネージャは，全てのロードされたモジュールのリストを 1 保持し，それぞれのリファレンスのリストを管理します．Mosel には，ユーザが，どのモジュールがいつロードされたか (XPRMgetnextdso)，提供されている特徴は何か (XXRPXgetdsoinfo) を知ったり，制御パラメータの値にアクセスする (XPRMgetnextdsoparam)

ための関数があります．XPRMpreloaddso と XPRMflushdso を用いて，モジュールはモデルファイルとは独立にロードしたり，アンロードしたりできます．

モデルコンパイラライブラリ Mosel モデルコンパイラライブラリ (xprm_mc) に含まれる関数は，XPRMcompmod ただひとつです．この関数を用いて，ライブラリユーザはモデルをロードしたり実行したりする前に Mosel ファイルをコンパイルします．xprm_mc は，ランタイムライブラリ xprm_rt を必要とします．関数 XPRMcompmod が関数 XPRMinit で Mosel を初期化するためだけに使われます．

3.4 クイック・リファレンス

初期化と終了

XPRMinit	Mosel ライブラリを初期化する．
XPRMfree	Mosel が用いているメモリを開放する．
XPRMgetversion	Mosel のバージョン番号を返す．

モデル管理

XPRMloadmod	Mosel にモデルをロードする．
XPRMrunmod	現在ロードされているモデルを実行する．
XPRMisrunmod	現在モデルが実行されているかどうかをチェックする．
XPRMstoprunmod	現在実行されているモデルを停止する．
XPRMunloadmod	Mosel からモデルをアンロードする．
XPRMgetmodinfo	モデルに関する情報を返す．
XPRMgetnextmod	次のモデルへの問題ポインタを返す．
XPRMfindmod	現在ロードされているモデルを見つける．

処理後のインタフェース

XPRMfindident	ディレクトリの識別子を見つける．
XPRMgetnextident	記号の表の次の識別子を返す．
XPRMgetnextparam	次のモデルパラメータを返す．
XPRMgetnextproc	手続き/関数の次のバージョンを返す．
XPRMgetprocinfo	手続き/関数に関する情報を提供する．

集合

XPRMgetsetsize	集合の大きさを返す．
XPRMgetsettype	集合の型を返す．
XPRMfirstsetndx	集合の最初の要素の添字を返す．
XPRMlastsetndx	集合の最後の要素の添字を返す．
XPRMgetelsetval	集合の要素の値を返す．
XPRMgetelsetndx	集合の要素の添字を返す．

配列

XPRMgetarrdim	配列の次元を返す.
XPRMgetarrtype	配列の型を返す.
XPRMgetarrsize	配列の大きさを返す.
XPRMgetarrsets	配列の添字集合を探す.
XPRMfirstarrentry	配列の最初のエントリの添字を探す.
XPRMlastarrentry	配列の最後のエントリの添字を探す.
XPRMnextarrentry	配列の次のエントリの添字を探す.
XPRMnextarrentry_trs	転置された配列の添字を探す.
XPRMfirstarrtruenty	最初の真の配列のエントリの添字のリストを探す.
XPRMnextarrtruentry	最後の真の配列のエントリの添字のリストを探す.
XPRMchkarrind	添字が配列に適切かどうかをチェックする.
XPRMcmpindices	2 つの添字を比較する.
XPRMgetarrval	配列のエントリの値を返す.

問題

XPRMgetprobstat	モデルの問題の状態を返す.
XPRMexportprob	スクリーンかファイルに行列を書く.
XPRMgetobjval	目的関数値を返す.
XPRMgetobjval	決定変数の解の値を返す.
XPRMgetcsol	制約式の値を返す.
XPRMgetrcost	リデューストコストを返す.
XPRMgetdual	制約式の双対値を返す.
XPRMgetslack	制約式のスラック値を返す.
XPRMgetact	制約式のアクティビティ値を返す.
XPRMgetvarnum	決定変数の列数を返す.
XPRMgetctrnum	制約式の行数を返す.

モジュールの取り扱い

XPRMautounloaddso	DSO の自動的なアンロードを可能/不可能にする.
XPRMfinddso	モジュール名から DSO ディスクリプタを返す.
XPRMflushdso	使っていない DSO をアンロードする.
XPRMgetdsoparam	制御/属性の現在の値を返す.
XPRMgetnextdso	次の DSO を返す.
XPRMgetnextdsoconst	DSO の次の制約式を返す.
XPRMgetnextdsoparam	DSO の次のパラメータを返す.
XPRMgetnextdsoproc	DSO の次の関数/手続きを返す.
XPRMgetdsoinfo	DSO についての情報を探す.
XPRMpreloaddso	明示的に DSO をロードする.

モデルコンパイラ

`XPRMcompmod` Mosel モデルをコンパイルし，BIM ファイルを作る．

第 4 章

Moselの関数と手続き

4.1 言語リファレンス

Mosel 言語は，2 章「Mosel 言語」に示したように，ブロックと命令のリストと，あらかじめ定義された関数と手続きから成ります．この章では，全ての関数と手続きの使い方のリファレンスを，例と使うときに注意すべき点とともに述べます．

コマンドのリストは Mosel 言語コアに使われるものだけではなく，言語を拡張するためのさまざまなモジュールのものを含み，Mosel のユーザに「ワンステップ・リファレンス」を提供します．このアプローチは，言語の開発を反映するというよりも，使い方を間違えないようにするためのコマンドの説明をめざしているものです．

ある関数や手続きは，入力としてあらかじめ定義された定数を採用したり，出力としてあらかじめ定義された定数に対応する値を返したりします．全ての場合において，これらの定数は関数や手続きと一緒に説明してあります．それに加えて，Mosel には次のような便利な数学的定数もあります．

`MAX_INT` integer の最大値

`MAX_REAL` real の最大値

`M_E` 自然対数の底 e

`M_PI` π

4.2 説明のためのレイアウト

本章で説明されている関数と手続きは，次のような項目に沿っています．

目的 コマンドの短い説明です．ここから情報を始めるという目的です．

概略 コマンドの使い方の文法の概略です．関数/手続きの使い方がいくつかの形式で書かれています．

引数 コマンドの引数のリストです．

モジュール 本章のコマンドは，Mosel 言語コアに含まれるものだけではありません．他のモジュールに含まれる関数は，モジュールの名前が書いてあります．このようなコマンドは，モデルファイルの一番上で以下のような行を書く必要があります．

`uses` *"module_name"*

関連する制御 コマンドに影響のある制御を，型ごとに分けて列挙してあります．これらの制御はコマンドを呼び出す前にセットしておく必要があります.

例 1 つか 2 つのコマンド使用例です.

補足 その他の情報です.

関連するトピック コマンドに関連するトピックや比較や参照のためのトピックです.

abs

目的 integer と real の絶対値を返す.

概略
```
function abs(i: integer): integer
function abs(r: real): real
```

引数

i	絶対値を計算する整数値.
r	絶対値を計算する実数値.

関連する制御 なし.

例 入力された数字が非負かどうかをチェックする.

```
declarations
  number: real
end-declarations
writeln("Enter a non-negative number.")
readln(number)
if(number <> abs(number)) then
  writeln("The number is negative.")
end-if
```

関連するトピック exp,ln,log,sqrt

addcut

目的 オプティマイザにおいて，問題にカットを加える．

概略 `procedure addcut(cuttype: integer, type: integer,linexp: linctr)`

引数

`cuttype`	カットを識別するための整数．
`type`	カットの型．以下のうちのひとつ．
	`CT_GEQ` 不均等（同じかそれより大きい）
	`CT_LEQ` 不均等（同じかそれより小さい）
	`CT_EQ` 均等
`linexp`	線形式．(値をとる範囲が決まっていない制約式)

モジュール `mmxprs`

関連する制御 *Boolean*

`VERBOSE` オプティマイザによって表示されるメッセージを有効/無効にする．

補足 これは，オプティマイザにおいて，カットを問題に加える手続きです．カットは現在のノードとそれから派生する全てのノードに適用されます．

関連するトピック `addcuts`，`dropcuts`，`delcuts`，`loadcuts`，`storecut`，`storecuts`，`XPRSaddcuts` (オプティマイザリファレンスマニュアル参照)

addcuts

目的 オプティマイザにおいて，問題にカットの配列を加える．

概略
```
procedure addcuts(cuttype: array(range) of integer,
    type: array(range) of integer,
    linexp: array(range) of linctr)
```

引数

cuttype	カットを識別するための整数の配列.
type	カットの型の配列．以下のうちのひとつ.
	CT_GEQ 不均等（同じかそれより大きい）
	CT_LEQ 不均等（同じかそれより小さい）
	CT_EQ 均等
linexp	線形式の配列.（値をとる範囲が決まっていない制約式）

モジュール mmxprs

関連する制御 *Boolean*

VERBOSE オプティマイザによって表示されるメッセージを有効/無効にする．

補足 これは，オプティマイザにおいて，カットの配列を問題に加える手続きです．カットは現在のノードとそれから派生する全てのノードに適用されます．この手続きのパラメータである 3 つの配列は同じ添字集合を持っている必要があることに注意しなければなりません．

関連するトピック addcut，dropcuts，delcuts，loadcuts，storecut，storecuts，XPRSaddcuts(オプティマイザリファレンスマニュアル参照)

arctan

目的 値のアークタンジェント（逆正接）を返す.

概略 `function arctan(r: real): real`

引数

`r`	三角関数に与える実数値.

関連する制御 なし

例 以下では，2 つの実数値 a と b を読み，それらを直角三角形の 2 つの辺であるとして，それらの辺の間の角度を「°」の単位で計算している.

```
declarations
  a,b: real
end-declaration
read(a,b)
write("The triangle has angles ",arctan(a/b)*180/M_PI)
writeln(", ",arctan(b/a)*180/M_PI," and 90 degrees")
```

関連するトピック cos,sin

bittest

目的 ビットセットをテストし，マスクで選ばれたビットを返す.

概略 `function bittest(i: integer, mask: integer): integer`

引数

`i`	テストされる非負の整数値.
`mask`	ビットマスク.

関連する制御 なし.

例 以下では，`i` の値が 4，`j` の値は 5，`k` の値は 8 となる.

```
i := bittest(12,5)
j := bittest(13,5)
k := bittest(13,10)
```

補足 この関数は，与えられた数をビットマスクと比較し，マスクによって選ばれたビットを選んで返します.（0 ビットは値 1，1 ビットは値 2，2 ビットは値 4 などである.）

関連するトピック なし.

ceil

目的 値を次に大きい整数に切り上げる.

概略 `function ceil(r: real): integer`

引数

`r`	値を丸める実数値.

関連する制御 なし.

例 以下では，`d` の値は-1 である.

```
d := ceil(-1.9)
```

関連するトピック `floor,round`

clearmipdir

目的 定義されている全ての MIP 命令を削除する.

概略 `procedure clearmipdir`

引数 なし.

モジュール `mmxprs`

関連する制御 *Boolean*

`VERBOSE` オプティマイザによって表示されるメッセージを有効/無効にする.

関連するトピック `setmipdir`, `XPRSgetdirs`(オプティマイザリファレンスマニュアル参照) ,`XPRSloaddirs`(オプティマイザリファレンスマニュアル参照)

clearmodcut

目的 定義されている全てのモデルのカットを削除する.

概略 `procedure clearmodcut`

引数 なし.

モジュール `mmxprs`

関連する制御 *Boolean*

VERBOSE オプティマイザによって表示されるメッセージを有効/無効にする.

例 以下では，問題を解く前に，モデルのカットとして制約式をセットしている．最後に，モデルのカットが削除されている.

```
declarations
  Items: set of integer
  ctr: array(Items) of linctr
end-declarations
   ...
forall(i in Items|isodd(i))
  setmodcut(ctr(i))
maximize(profit)
writeln("Profit: ",getobjval)
clearmodcut
```

関連するトピック `setmodcut`

cos

目的 値のコサイン（余弦）を返す.

概略 `function cos(r: real): real`

引数

`r`	三角関数に与える実数値.

関連する制御 なし.

例 以下では，三角形の 2 辺とその間の角度を読み，もうひとつの辺の長さを計算している.

```
declarations
  b,c,A: real
end-declarations

read(b,c,A)
write("The other side has length ")
writeln(sqrt(b*b + c*c -2*b*c*cos(A*M_PI/180)))
```

関連するトピック `arctan,sin`

create

目的 既に宣言されている動的な配列の一部として決定変数を作る.

概略 `procedure create(x: mpvar)`

引数

`x`	作りたい変数.

関連する制御 なし.

例 以下では，変数の動的な配列を宣言し，奇数の添字に対応する変数のみを生成する．最後に，一次式 `x(1) + x(3) + x(5) + x(7)` を定義している.

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

forall(i in 1..8| isodd(i)) create(x(i))

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

補足 配列の変数が動的に宣言されている場合（または添字が動的な集合である場合），要素は宣言されたときには作られません．この手続きを使って作る必要があります.

関連するトピック `finalize`(2 章「Mosel 言語」の「配列」参照)

delbasis

目的 手続き savebasis で保存された基底を削除する.

概略 `procedure delbasis(num: integer)`
`procedure delbasis(name: string)`

引数

num	保存されている基底のリファレンス番号.
name	保存されている基底の名前.

モジュール mmxprs

関連する制御 *Boolean*
VERBOSE オプティマイザによって表示されるメッセージを有効/無効にする.

例 以下では，リファレンス番号を 2 にして基底を保存し，ほかのことを実行してから最後にリファレンス番号 2 の基底を削除している.

```
savebasis(2)
  ...
delbasis(2)
```

関連するトピック loadbasis，savebasis，XPRSgetbasis (オプティマイザリファレンスマニュアル参照) ,XPRSloadbasis(オプティマイザリファレンスマニュアル参照)

delcuts

目的 オプティマイザにおいて，問題からカットを削除する．

概略
```
procedure delcuts(keepbasis: boolean, cuttype: integer,
  interpret: integer, delta: real, cuts: set of integer)
procedure delcuts(keepbasis: boolean, cuttype: integer,
  interpret: integer, delta: real)
```

引数

keepbasis		false 基本的なスラックではないカットが削除される．
		true 基底が正しいことを保証する．
cuttype		カットを識別するための整数．
interpret		カットの型がインタープリットされる方法．
	-1	全てのカットを削除．
	1	カットの型を数として扱う．
	2	カットの型をビットマップとして扱う．（cuttype のビットセットのどれかにマッチするカットを削除．）
	3	カットの型をビットマップとして扱う．（cuttype のビットセットの全てにマッチするカットを削除．）
delta		スラック値の絶対値が delta より大きいカットだけを削除する．全てのカットを削除するには，この引数を小さい値（-MAX_REAL）にしておけばよい．
cuts		カットの添字集合．指定されない場合，型 cuttype の全てのカットが削除される．

モジュール mmxprs

関連する制御 *Boolean*

VERBOSE オプティマイザによって表示されるメッセージを有効/無効にする．

補足 この手続きは，オプティマイザにロードされている問題からカットを削除します．カットがある基準によって除外されている場合には削除されません．

関連するトピック addcut，addcuts，dropcuts，loadcuts，storecut，storecuts，XPRSdelcuts (オプティマイザリファレンスマニュアル参照)

dropcuts

目的 カットプールからカットの集合を削除する.

概略
```
procedure dropcuts(cuttype: integer, interpret: integer,
cuts: set of integer)
procedure dropcuts(cuttype: integer, interpret: integer)
```

引数

cuttype		カットを識別するための整数.
interpret		カットの型がインタープリットされる方法.
	-1	全てのカットを削除.
	1	カットの型を数として扱う.
	2	カットの型をビットマップとして扱う.(cuttype のビットセットのどれかにマッチするカットを削除.)
	3	カットの型をビットマップとして扱う.(cuttype のビットセットの全てにマッチするカットを削除.)
cuts		カットプールにあるカットの添字集合. 指定されない場合, 型 cuttype の全てのカットが削除される.

モジュール mmxprs

関連する制御 *Boolean*
VERBOSE オプティマイザによって表示されるメッセージを有効/無効にする.

補足 この手続きは, カットプールからカットの集合を削除します. アクティブなノードに適用されていないカットだけが削除されます.

関連するトピック addcut, addcuts, delcuts, loadcuts, storecut, storecuts, XPRSdelcpcuts(オプティマイザリファレンスマニュアル参照)

exlsts

目的 動的な配列にエントリが作られたかどうかをチェックする.

概略 `function exists(x): boolean`

引数

`x`	配列の参照 (例えば，`t(1)`)

関連する制御 なし.

例 以下の例では，決定変数の動的な配列の偶数の要素だけが作られている．存在している変数を表示している.

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

forall(i in 1..8|isodd(i)=false) create(x(i))

c:= sum(i in 1..8) x(i)

forall(i in 1..8)
  if(exists(x(i)) = true) then
    writeln("x(",i,") exists")
  end-if
```

補足 配列の変数が動的に宣言されている場合（または添字が動的な集合である場合），要素は宣言されたときには作られません．この関数は，与えられた要素が作られているかどうかを示し，作られているなら真，そうでなければ偽を返します.

関連するトピック `create`

exit

目的 現在のプログラムを中止し，値を返す.

概略 `procedure exit(code: integer)`

引数

`code`	プログラムから返される値.

関連する制御 なし.

例 以下では，対数を計算したい値を読む．数が正でない場合には，中止して 1 を返す.

```
readln(number)

if(number > 0) then
  writeln(log(number))
else
  exit(1)
end-if
```

補足 モデルは，`exit` を使って変更されなければ，デフォルトでは 0 を返して終了します.

関連するトピック `QUIT`

exp

目的 値のエクスポーネント（指数）を返す.

概略 `function exp(r: real): real`

引数

`r`	関数に適用する実数値.

関連する制御 なし.

例 下に示す 2 つの関数を使って指数の値を計算し，関数 `exp` で計算した値と比較する.

```
declarations
  limit = 100
  xp = 0
end-declarations

read(number)

forall(i in 0..limit) do
  xp := xp + pospow(number,limit-i)/factorial(limit-i)
end-do

writeln("Calculated: ",xp,"; exp(",number,"): ",
      exp(number))
```

使っている関数は以下の通り.

```
function factorial(a: integer): real
  if(a < 0) then
    returned := -1
  else if(a = 1 or a = 0) then
    returned := 1
    else
      returned := a*factorial(a-1)
    end-if
  end-if
end-function

function pospow(x:real, e:integer): real
  if(e<=0) then
```

```
    returned := 1
  else
    returned := x*pospow(x,e-1)
  end-if
end-function
```

関連するトピック abs,ln.log,sqrt

exportprob

目的 問題をファイルに出力する.

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

引数

options		ファイル形式のオプション.
	EP_MIN	LP 形式，最小化（デフォルト）.
	EP_MAX	LP 形式，最大化.
	EP_MPS	MPS 形式.
	EP_STRIP	スクランブルされた名前を使う.
filename		出力ファイルの名前. 空の文字列が与えられた場合には標準出力（スクリーン）に表示.
obj		目的関数制約.

関連する制御 なし.

例 以下では，最初に問題がスクリーンに表示され，LP 形式，最大化で，スクランブルされた名前をファイルに prob1.lp 出力する.

```
Profit:
  declarations
    MinCost,Profit: linctr
  end-declarations

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

補足 拡張子が与えられない場合には，LP 形式なら.lp，MPS 形式なら.mps という拡張子が自動的につけられます.

関連するトピック XPRMexportprob (EXPORTPROB), XPRBexportprob

fclose

目的 アクティブな入力または出力ストリームを閉じる.

概略 `procedure fclose(stream: integer)`

引数

`stream`	閉じるストリーム. `F_INPUT` 入力ストリーム `F_OUTPUT` 出力ストリーム

関連する制御 *Boolean*

`IOCTRL` インタープリタが IO エラーを無視する/しない.

例 以下では，最適化した後，解のデータがファイルに書かれている.

```
declarations
  profit: linctr
  variables: set of mpvar
end-declarations
   ...
fopen("Solution.dat",F_OUTPUT)
  writeln("Profit: ",getobjval)
  getvars(profit,variables)
  forall(x in variables) writeln(getsol(x))
fclose(F_OUTPUT)
```

補足 この手続きは，与えられたストリームの現在のファイルを閉じます．一つ前に開かれたファイルが次に対応するストリームになります．ファイルを閉じるには `fopen` で開いてある必要があります．対応するストリームがない場合や明示的に開かれたファイルがない場合には何もしません.

関連するトピック `fopen`, `fselect`, `getfid`, `iseof`, `SQLconnect`, `SQLdisconnect`.

delete

目的 ファイルを削除する.

概略 `procedure fdelete(filename: string)`

引数

`filename`	削除するファイルの名前とパス.

モジュール `mmsystem`

関連する制御 なし.

例 以下では，ログファイルを明示的に削除して，その後，新しく出力ファイルを開いている.

```
fdelete("logfile.log")
fopen("logfile.log",F_OUTPUT)
    ...
fclose(F_OUTPUT)
```

関連するトピック `fmove`, `getfstat`

fflush

目的 OS を用いてデフォルトの出力ストリームのバッファが有するデータを書き出す.

概略 `procedure fflush`

引数 なし.

関連する制御 なし.

関連するトピック `XPRMflushdso (FLUSHLIBS)`

finalize

目的 集合の定義を確定する.

概略 `procedure finalize(s: set)`

引数

`s`	動的な集合.

関連する制御 なし.

例 以下では，添字集合が定義されており，決定変数の動的な配列によっている．集合は最終的に 3 つの要素を持つと定義され，この大きさで確定されている．このとき，大きさの固定された配列が出力される.

```
declarations
  Set1: set of string
  x: array(Set1) of mpvar
end-declarations

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

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

補足 この手続きは集合の定義を確定します．これによって，現在その集合に含まれる要素からなる動的な集合が大きさの決まった集合になります．したがって，この集合が添字となる配列は，静的な（大きさが固定の）配列になります．集合が確定される前に宣言された配列は状態は動的ですが，要素は変更できません.

関連するトピック `create`

floor

目的 値を次に小さい整数に切り下げる.

概略 `function floor(r: real): integer`

引数

`r`	値を丸める実数値.

関連する制御 なし.

例 以下では，`i` の値は 5，`j` の値は-7，`k` の値は 12 である.

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

関連するトピック `ceil,round`

fmove

目的 ファイルの名前を変えるか移動する.

概略 `procedure fmove(namesrc: string, namedest: string)`

引数

namesrc	名前を変えるか移動するファイルの名前とパス.
namedest	行き先の名前と/またはパス.

モジュール mmsystem

関連する制御 なし.

例 以下では，モデルを実行したときのログファイルを，実行した順番に番号をつけたファイルに移動している.

```
forall(i in 1..100) do
  file := "logfile" + i
  status := getfstat(file)
  if(bittest(status,SYS_TYP) = SYS_REG) then next
  else break
  end-if
end-do

fmove("logfile.log",file)
fopen("logfile.log",F_OUTPUT)
    ...
```

補足 この手続きは，namesrc の名前を namedest に変えます．2 番目の名前がディレクトリなら，そのディレクトリに移動します.

関連するトピック fdelete, getfstat

fopen

目的 ファイルを開き，アクティブな入力または出力ストリームにする．

概略 `procedure fopen(f: string, mode: integer)`

引数

`f`	開くファイルの名前.
`mode`	開くモード.
	`F_INPUT` 読み込むために開く.
	`F_OUTPUT` 書き込むために空のファイルを開く.
	`F_APPEND` 書き込むためにファイルを開き，ファイルの最後に追加する.

関連する制御 *Boolean*

`IOCTRL` インタープリタが IO エラーを無視する/しない.

例 以下では，目的関数の値がデータファイルに追加する．

```
fopen("Data.dat",F_APPEND)
if(getparam(IOSTATUS) <> 0) then
  exit(1)
end-if
writeln(getobjval)
fclose(F_OUTPUT)
```

補足

1. この手続きは，読み書きするためにファイルを開きます．成功すると，モードによってファイルが（デフォルトでは）アクティブな入力または出力ストリームになります．手続き `write` と `writeln` はデフォルトの出力ストリームにデータを書くために用い，また，関数 `read`，`readln` と `fskipline` はデフォルトの入力ストリームからデータを読むために用います．
2. IO エラーの時（ファイルが開けないなど）には，制御パラメータ `IOCTRL` によって動作が決まります．この値が `false` の場合（デフォルトは `false`）にはインタープリタが停止します．そうでない場合にはエラーを無視して続けます．IO エラーの状態は属性 `LOSTATUS` に格納されます．実行に成功すると 0 になります．このパラメータは関数 `getparam` を使って読まれるとリセットされることに注意してください．

関連するトピック `fclose`，`fselect`，`getfid`，`iseof`，`SQLconnect`，`SQLdisconnect`

fselect

目的 与えられたストリームをアクティブな入力または出力ストリームにする.

概略 `procedure fselect(stream: integer)`

引数

`stream`	ストリーム番号.

関連する制御 なし.

例 以下では，デフォルトの出力ストリームのファイル ID を保存し，出力ストリームを `mylog.txt` に変える．現在の出力ストリームのファイル ID を保存して再びデフォルトの出力を選ぶ.

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

補足 ストリームを開いたときの状態によって，入力または出力ストリームのどちらに割り当てられるか決まります．ストリーム番号は関数 `getfid` によって得ることができます.

関連するトピック `fclose`, `fopen`, `getfid`, `iseof`

fskipline

目的 デフォルトの入力ストリームのコメント行を読み飛ばす.

概略 `procedure fskipline(filter: string)`

引数

`filter`	コメントの記号のリスト

関連する制御 *Boolean*

IOCTRL インタープリタが IO エラーを無視する/しない.

例 以下では,「#」と「!」で始まる行と空行をスキップする.

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

補足 この手続きは，入力ストリームのコメント行を，コメント記号をフィルタにして飛ばします．与えられた記号は，コメント行の始まりと考えられますす．文字「¥n」を指定したときには空行を示すことに注してください．パースでは，スペースとタブは無視されます.

関連するトピック `read,readln`

getact

目的 問題を解くのに成功した場合には，制約のアクティビティ値を返す．そうでない場合は 0 を返す．

概略 `function getact(c: linctr): real`

引数

`c` 線形制約式．

関連する制御 なし．

例 制約式のアクティビティ値とスラック値の和は右辺の値と一致するので，解の簡単なチェックができる．この例はこれを行う方法を示す．

```
function check(ctr: linctr, name: string): boolean
  if((gettype(ctr)=CT_GEQ or gettype(ctr)=CT_LEQ) and
      (getact(ctr)+getslack(ctr)+getcoeff(ctr)<>0))
  then
    writeln("Error in constraint '",name,"'")
    returned := false
  end-if
  returned := true
end-function
```

この関数は以下のように使用する．

```
maximize(profit)
if(check(first,"first") and check(second,"second")) then
  writeln("Profit: ",getobjval)
end-if
```

関連するトピック `getdual`, `getslack`, `getsol`, `XPRMgetact`

getcnlist

目的 現在のノードでアクティブなカットの集合を得る.

概略 `procedure getcnlist(cuttype: integer, interpret: integer,`
`cuts: set of integer)`

引数

cuttype		カットを識別するための整数. 全てのアクティブなカットの場合には-1 とする.
interpret		カットの型がインタープリットされる方法.
	-1	全てのカットを得る.
	1	カットの型を数として扱う.
	2	カットの型をビットマップとして扱う.(cuttype のビットセットのどれかにマッチするカットを得る.)
	3	カットの型をビットマップとして扱う.(cuttype のビットセットの全てにマッチするカットを得る.)
cuts		カットの添字集合.

モジュール mmxprs

関連する制御 *Boolean*
VERBOSE オプティマイザによって表示されるメッセージを有効/無効にする.

補足 この手続きは, オプティマイザにおいて, 現在のノードのアクティブなカットの集合を得るものです. カットの添字集合は, パラメータ cuts に返されます.

関連するトピック getcplist, XPRSgetcutlist(オプティマイザリファレンスマニュアル参照)

getcoeff

目的 与えられた制約式の変数の係数を返す．変数が与えられないときには制約式の定数項（=-RHS）を返す．

概略
```
function getcoeff(c: linctr): real
function getcoeff(c: linctr, x: mpvar): real
```

引数

c	線形制約式．
x	決定変数．

関連する制御 なし．

例 この例では，3 つの変数をもつひとつの制約式が定義されている．getcoeff の結果は，r が -1.0，s が -12.0 である．

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

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

補足 返り値は，制約式の正規化された表現に対応します．全ての変数と定数項は等号（不等号）の左にあります．

関連するトピック getvars，setcoeff，XPRSgetrows (オプティマイザリファレンスマニュアル参照)

getcplist

目的 カットプールからカットの添字集合を得る.

概略
```
procedure getcplist(cuttype: integer, interpret: integer,
    delta: real, cuts: set of integer,
    viol: array(range) of real)
```

引数

cuttype		カットを識別するための整数.
interpret		カットの型がインタープリットされる方法.
	-1	全てのカットを得る.
	1	カットの型を数として扱う.
	2	カットの型をビットマップとして扱う.(cuttype のビットセットのどれかにマッチするカットを得る.)
	3	カットの型をビットマップとして扱う.(cuttype のビットセットの全てにマッチするカットを得る.)
delta		スラック値の絶対値が delta より大きいカットだけを得る.
cuts		カットプールでのカットの添字集合.
viol		カットのスラック値が返される配列.

モジュール mmxprs

関連する制御 *Boolean*

VERBOSE オプティマイザによって表示されるメッセージを有効/無効にする.

補足 この手続きではカットプールからカットの添字集合を得ます. パラメータ cuts に返されます.

関連するトピック getcnlist, XPRSgetcpcutlist(オプティマイザリファレンスマニュアル参照) ,XPRSgetcpcuts(オプティマイザリファレンスマニュアル参照)

getcwd

目的 現在の作業ディレクトリを返す.

概略 `function getcwd: string`

引数 なし.

モジュール `mmsystem`

関連する制御 なし.

例 以下では，現在の作業ディレクトリを表示している.

```
procedure printdir
  writeln("Executing from ",getcwd)
end-procedure
```

補足 この関数は現在の作業ディレクトリを返します．Mosel が実行され，ファイルを探すディレクトリです.

関連するトピック `getenv`，`getfstat`，`makedir`，`removedir`

getdual

目的 解くのに成功した問題の制約式の双対値を返す．そうでない場合は 0 を返す．

概略 `function getdual(c: linctr): real`

引数

`c` 線形制約式．

関連する制御 なし．

例 以下では，問題を解き，制約式のひとつの双対値を見つけている．

```
first := 3*a + 2*b <= 400
second := a + 3*b <= 200
profit := a + 2*b
maximize(profit)
writeln("Dual value of constraint first is ",
      getdual(first));
```

関連するトピック `getobjval`，`getrcost`，`getslack`，`getsol`，`XPRMgetdual`，`XPRSgetsol` (オプティマイザリファレンスマニュアル参照)

getenv

目的 OS の環境変数の値を返す.

概略 `function getenv(name: string): string`

引数

`name`	環境変数の名前.

モジュール `mmsystem`

関連する制御 なし.

例 以下では，環境変数 PATH の値を返している.

```
str := getenv("PATH")
```

関連するトピック `getcwd`, `getsysstat`, `system`

getfid

目的 アクティブな入力または出力ストリームのストリーム番号を返す.

概略 `function getfid(stream: integer): integer`

引数

`stream`	参照するストリーム. `F_INPUT` 入力ストリーム `F_OUTPUT` 出力ストリーム

関連する制御 なし.

例 以下では，標準ストリームのストリーム番号が変数 `def_out` に保存され，新しい出力ストリーム `file` が開かれている．目的関数の値が `file` に送られたら，ストリーム番号が保存され，また標準出力に戻っている.

```
def_out := getfid(F_OUTPUT)
fopen("file",F_OUTPUT)
writeln("Objective function value: ",getobjval)
my_out := getfid(F_OUTPUT)
fselect(def_out)
```

補足 返り値は関数 `fselect` へのパラメータとして使うことができます.

関連するトピック `fclose`, `fopen`, `fselect`, `iseof`

getfirst

目的 範囲集合の最初のエレメントを返す．空なら 0 を返す．

概略 `function getfirst(r: range): integer`

引数

`r`	範囲集合．

関連する制御 なし．

例 以下の例では，範囲集合 `r` が定義されて最初と最後のエレメントが表示されている．

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

関連するトピック `getlast`

getfstat

目的 ファイルとディレクトリの状態（ビットエンコードされた型とアクセスモード）を返す.

概略 `function getfstat(filename: string): integer`

引数

`filename` チェックするファイルかディレクトリ名前（またはパス）.

モジュール `mmsystem`

関連する制御 なし.

例 以下では，`ftest` がディレクトリかどうかと書き込み可能かどうかを調べている.

```
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
```

補足 状態の型は定数マスク SYS_TYP(排他) を使ってデコードすることができます. 可能な値は以下の通りです.

SYS_DIR ディレクトリ
SYS_REG 普通のファイル.
SYS_OTH 特別なファイル（デバイス，パイプなど）

アクセスモードは定数マスク SYS_MOD(和) でデコードすることができます. 可能な値は以下の通りです.

SYS_READ 読むことができる.
SYS_WRITE 変更ができる.
SYS_EXEC 実行できる.

関連するトピック `bittest`

getlast

目的 範囲集合の最後の要素を返す．空なら 0 を返す．

概略 `function getlast(r: range): integer`

引数

`r` 範囲集合．

関連する制御 なし．

例 以下の例では，範囲集合 `r` が定義されて最初と最後のエレメントが表示されている．

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

関連するトピック `getfirst`

getlb

目的 変数の最小値を返す.

概略 `function getlb(x: mpvar): real`

引数

`x` 決定変数.

モジュール `mmxprs`

関連する制御 *Boolean*

`VERBOSE` オプティマイザによって表示されるメッセージを有効/無効にする.

例 以下では，配列 `x` の全ての変数の最小値と最大値が表示されている.

```
declarations
  Items = 1..100
  x: array(Items) of mpvar
end-declarations
   ...
forall(i in Items)
  writeln(i,": ",getlb(x(i)),", ",getub(x(i)))
```

補足 この関数は，オプティマイザによって現在保持されている変数の最小値を返します．関数 `setlb` を使ってオプティマイザの中で最小値を直接変更できます．Mosel での変数の変化は，問題がリロード（手続き `loadprob`）されない限り，この関数には影響しません.

関連するトピック `getub`，`setlb`，`setub`，`XPRSchgbounds` (オプティマイザリファレンスマニュアル参照) ,`XPRSgetlb` (オプティマイザリファレンスマニュアル参照)

getobjval

目的 問題を解くのに成功した場合には目的関数の値を返す．そうでなければ 0 を返す．

概略 `function getobjval: real`

引数 なし．

関連する制御 なし．

例 この例では，目的関数 `profit` が最大化されてその値が返されている．

```
maximize(profit)
writeln("Maximum profit obtainable is ",getobjval)
```

補足 整数解が得られている場合には，最適値が返されます．そうでない場合には，最後の LP 解が返されます．

関連するトピック `getdual`，`getprobstat`，`getrcost`，`getslack`，`getsol`，`LPOBJVAL` (オプティマイザリファレンスマニュアル参照) ,`MIPOBJVAL`(オプティマイザリファレンスマニュアル参照)

getparam

目的 制御パラメータまたは問題属性の現在の値を返す.

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

引数

`name`	値を求める制御パラメータまたは問題属性（大文字小文字を区別する）. 制御のリストは 6 章「制御パラメータ」, 属性のリストは 7 章「問題属性」を参照.

関連する制御 6 章「制御パラメータ」参照.

例 以下では, 最後に実行されて成功した SQL コマンドを決定している.

```
success := getparam("SQLSUCCESS")
```

補足 このコマンドで値が返されるパラメータは, Mosel とロードされているモジュールに含まれるコマンドによってセットされます. モジュールの名前は名前にモジュールの名前を前置して特定します（例えば,「`mmxprs.XPRS_VERBOSE`」）. 返り値の型はパラメータの型に依存します.

関連するトピック `setparam`(オプティマイザリファレンスマニュアル 7 章「制御パラメータ」と 8 章「問題属性」参照)

getprobstat

目的 オプティマイザの問題の状態を返す.

概略 `function getprobstat: integer`

引数 なし.

モジュール `mmxprs`

関連する制御 *Boolean*

VERBOSE オプティマイザによって表示されるメッセージを有効/無効にする.

例 以下では，getobjval を呼び出す前に，問題の状態を調べて解が得られるかどうかを決定している.

```
maximize(profit)
if(getprobstat = XPRS_OPT) then
  writeln("Maximum profit is ",getobjval)
else
  writeln("Problem could not be solved to optimality")
end-if
```

補足 getprobstat の返り値はオプティマイザに保持されている問題の現在の状態で，以下のような値をとり得ます.

XPRS_OPT　最適解が得られた.
XPRS_UNF　終わっていない.
XPRS_INF　実行できない.
XPRS_UNB　制約されていない.

この関数で得られるよりも詳細な情報は，関数 getparam で問題属性 PRESOLVESTATE，LPSTATUS，MIPSTATUS を調べることによって得られます. (オプティマイザリファレンスマニュアル参照)

関連するトピック XPRMgetprobstat，LPSTATUS (オプティマイザリファレンスマニュアル参照) ,MIPSTATUS (オプティマイザリファレンスマニュアル参照)

getrcost

目的 問題を解くのに成功し，その変数が問題に含まれている場合，削減されたコストの値を返す．そうでない場合は 0 を返す．

概略 `function getrcost(v: mpvar): real`

引数

v	決定変数.

関連する制御 なし．

例 以下では，最適解があるかどうかを調べ，その値と削減されたコストを表示する．

```
declarations
  profit: linctr
  variables: set of mpvar
end-declarations
    ...
if(getprobstat = XPRS_OPT) then
  getvars(profit,variables)
  forall(x in variables)
    writeln(getsol(x),"; ",getrcost(x))
  end-if
```

関連するトピック `getdual`, `getobjval`, `getprobstat`, `getslack`, `getsol`, `XPRMgetrcost`, `XPRMgetsol`(オプティマイザリファレンスマニュアル参照)

getsize

目的 配列または集合の大きさ（セルまたは要素の数）を返す.

概略
```
function getsize(a: array): integer
function getsize(s: set): integer
```

引数

a	配列.
s	集合.

関連する制御 なし.

例 以下では，動的な配列が宣言され，8 個の要素を保持し，2 つだけが実際には定義されている．getsize を呼び出して配列のサイズを調べると，その値は 2 である.

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

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

補足 配列が最大の範囲で動的に宣言されている場合，返り値は範囲よりも小さくなります.

関連するトピック XPRMgetarrsize，XPRMgetsetsize

getslack

目的 解くのに成功した問題の制約式のスラック値を返す．そうでない場合は 0 を返す．

概略 `function getslack(c: linctr): real`

引数

`c` 線形制約式．

関連する制御 なし．

例 以下では，問題の全てのスラック値を返している．

```
declarations
  ctr: set of linctr
end-declarations
   ...
if(getprobstat = XPRS_OPT) then
  forall(c in ctr)
    if(getslack(c) <> 0) then
      writeln(getslack(c))
    end-if
end-if
```

関連するトピック `getdual`, `getobjval`, `getprobstat`, `getrcost`, `getsol`, `XPRMgetslack`, `XPRSgetsol` (オプティマイザリファレンスマニュアル参照)

getsol

目的 問題を解くのに成功していて，変数が問題に含まれている場合には，変数の解の値を返す．そうでない場合は 0 を返す．制約式の場合には，現在の解を用いて対応する一次式の値を評価する．

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

引数

v	決定変数.
c	線形制約式.

関連する制御 なし．

例 以下では，最適解がチェックされ，全ての決定変数の解の値を出力ストリームに表示している．

```
declarations
  profit: linctr
  variables: set of mpvar
end-declarations
   ...
if(getprobstat = XPRS_OPT) then
  getvars(profit,variables)
  forall(x in variables)
    writeln(getsol(x))
end-if
```

関連するトピック getdual, getrcost, getobjval, getprobstat, XPRMgetcsol, XPRMgetvsol, XPRSgetsol (オプティマイザリファレンスマニュアル参照)

getsysstat

目的 システムの状態を返す.

概略 `function getsysstat: integer`

引数 なし.

モジュール `mmsystem`

関連する制御 なし.

例 この例では，ファイル `randomfile` を消去しようとしている．成功しなかった場合，メッセージが表示される.

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

補足 この関数はシステムの状態を返します．モジュールが最後に実行した動作が成功した場合には，返り値が 0 になります.

関連するトピック `getfstat`, `IOSTATUS`, `NBREAD`, `SQLSUCCESS`

gettime

目的 時間（秒単位）を返す.

概略 `function gettime: real`

引数 なし.

モジュール `mmsystem`

関連する制御 なし.

例 以下はプログラムの実行時間を表示している.

```
starttime := gettime
    ...
write("Time: ",gettime-starttime)
```

補足 絶対値はシステムに依存します.

関連するトピック なし.

gettype

目的 線形制約式の型を返す.

概略 `function gettype(c: linctr): integer`

引数

c	線形制約式.

関連する制御 なし.

例 この例では，最適解に続いて,「より小さい」と「より大きい」という制約式のスラック値が表示される.

```
declarations
c: set of linctr
end-declarations
    ...
if(getprobstat = XPRS_OPT) then
  forall(c in ctr)
  if(gettype(c) = CT_GEQ or gettype(c) = CT_LET) then
    writeln(getslack(c))
  end-if
end-if
```

補足 gettype は線形式の型を返します．可能な値は以下の通りです.

CT_EQ	等しい「=」.
CT_GEQ	同じかそれより大きい「≥」.
CT_LEQ	同じかそれより小さい「≤」.
CT_UNB	バインドされていない制約式.
CT_SOS1	型 1 の特殊順序集合.
CT_SOS2	型 2 の特殊順序集合.

単項制約式の値は以下の通りです.

CT_CONT	連続している.
CT_INT	整数である.
CT_BIN	バイナリである.
CT_PINT	整数の一部である.
CT_SEC	準連続である.
CT_SINT	準連続な整数である.

関連するトピック getdual, getslack, settype

getub

目的 変数の最大値を返す.

概略 `function getub(x: mpvar): real`

引数

`x` 決定変数.

モジュール `mmxprs`

関連する制御 *Boolean*

`VERBOSE` オプティマイザによって表示されるメッセージを有効/無効にする.

例 以下では，配列 `x` の全ての変数の最小値と最大値が表示されている.

```
declarations
  Items = 1..100
  x: array(Items) of mpvar
end-declarations
   ...
forall(i in Items)
  writeln(i,": ",getlb(x(i)),", ",getub(x(i)))
```

補足 この関数は，オプティマイザによって現在保持されている変数の最大値を返します．関数 `setub` を使ってオプティマイザの中で最小値を直接変更できます．Mosel での変数の変化は，問題がリロード（手続き `loadprob`）されない限り，この関数には影響しません.

関連するトピック `getlb`，`setlb`，`setub`，`XPRSchgbounds` (オプティマイザリファレンスマニュアル参照) ,`XPRSgetub` (オプティマイザリファレンスマニュアル参照)

getvars

目的 制約式の変数の集合を得る.

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

引数

`c`	線形制約式.
`s`	決定変数の集合を返すための引数.

関連する制御 なし.

例 以下では，集合 `vset` の線形制約式の変数の集合を返し，それら全ての解の値を見つけている.

```
declarations
  c: linctr
  vset: set of mpvar
end-declarations

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

関連するトピック `XPRMgetvarnum`, `COLS` (オプティマイザリファレンスマニュアル参照) , `XPRSgetcols` (オプティマイザリファレンスマニュアル参照)

initglobal

目的 大域的探索をリセットする.

概略 `procedure initglobal`

引数 なし.

モジュール `mmxprs`

関連する制御 なし.

補足 この手続きは，オプション `XPRS_NIG` をつきで `maxmize` または `minimize` によってスタートした大域的探索をリセットします.

関連するトピック `maximize,minimize`

iseof

目的 デフォルト入力ストリームの終わりかどうかをテストする.

概略 `function iseof: boolean`

引数 なし.

関連する制御 なし.

例 以下では，整数のデータファイルを開いて，ファイルの終わりまで行ごとに読んでコンソールに表示する.

```
declarations
  d: integer
end-declarations
   ...
fopen("datafile.dat",F_INPUT)
while(iseof = false) do
  readln(d)
  writeln(d)
end-do
fclose(F_INPUT)
```

関連するトピック `fclose`, `fopen`, `fselect`, `getfid`

ishidden

目的 制約式が隠されているかどうかをテストする.

概略 `function ishidden(c: linctr): boolean`

引数

`c` 線形制約式.

関連する制御 なし.

例 以下では，問題を解いて，隠されている制約式を戻して問題全体を解く.

```
declarations
  ctr: set of linctr
end-declarations
   ...
maximize(profit)
writeln(getobjval)
forall(c in ctr)
  if(ishidden(c)) then
    sethidden(c,false)
  end-if
maximize(profit)
writeln(getobjval)
```

補足 現在の問題に制約式が加えられるが，関数 `sethidden` を使えば隠すことができます．つまり，オプティマイザで解かれる制約式は問題に含まれなくなりますが，Mosel の問題の定義から削除されるわけではありません.

関連するトピック `sethidden`

isodd

目的 整数が奇数かどうかをテストする.

概略 `function isodd(i: integer): boolean`

引数

`i` 整数.

関連する制御 なし.

例 以下では，動的な変数の配列定義され，奇数の添字に対応する要素だけが作られている．最後に，一次式 `x(1) + x(3) + x(5) + x(7)` が定義される.

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

forall(i in 1..8| isodd(i)) create(x(i))

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

関連するトピック なし.

IVEaddtograph

目的 IVE ユーザグラフに点をプロットする.

概略 `procedure IVEaddtograph(xval: real, yval: real)`

引数

`xval`	点の x 軸の値.
`yval`	点の y 軸の値.

モジュール `mmive`

関連する制御 なし.

例 以下では，ランダムなグラフをプロットする.

```
IVEinitgraph("Random graph","xcoord","value")

forall (j in 1..20000 ) do
  IVEaddtograph(j,random)
end-do
```

補足

1. この関数は，ユーザグラフが最初に `IVEinitgraph`. を呼び出すことによって初期化した後に使えます.
2. 右に向かって 1 点ずつ点が順番にグラフに加えられます．これによって，複数の関数を書くことができます．前にプロットされたよりも x 軸の値が小さい点が加えられた場合には，違う関数が始まったと解釈されます.

関連するトピック `IVEinitgraph`

IVEinitgraph

目的 IVE でのグラフを初期化し，軸の名前をセットする．

概略 `procedure IVEinitgraph(name: string, x: string, y: string)`

引数

`name`	グラフの名前.
`x`	グラフの x 軸の名前.
`y`	グラフの y 軸の名前.

モジュール `mmive`

関連する制御 なし．

例 以下の例では，Xpress-MP エッセンシャルから，直接関数 $\exp(-0.01x)\cos(0.1x)$ の一部をプロットしている．

```
model decay
  uses "mmive";
  IVEinitgraph("Decaying oscillator","x","y")
  forall(i in 1..400)
    IVEaddtograph(i,exp(-0.01*i)*cos(i*0.1))
end-model
```

関連するトピック `IVEaddtograph`

ln

目的 自然対数の値を返す.

概略 `function ln(r: real): real`

引数

`r`	関数に与える実数値．正でなければならない.

関連する制御 なし.

例 1 以下では，`i` の値は 0，`j` は 1，`k` は数学的エラーである.

```
i := ln(1)
j := ln(M_E)
k := ln(0)
```

例 2 関数 `log` の例を参照.

関連するトピック `exp`，`log`，`sqrt`

loadbasis

目的 savebasis を使って保存されている基底を，オプティマイザにロードする．(delbasis を用いて) 削除されていなければ，この手続きを繰り返すことができる．

概略 `procedure loadbasis(num: integer)`
`procedure loadbasis(name: string)`

引数

num	保存されている基底のリファレンス番号.
name	保存されている基底の名前.

モジュール mmxprs

関連する制御 *Boolean*
VERBOSE オプティマイザによって表示されるメッセージを有効/無効にする．

例 以下では，リファレンス番号を 2 にして基底を保存し，ほかの基底をオプティマイザにロードし，古い基底をリロードする．

```
declarations
  MinCost: linctr
end-declarations

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

補足 オプティマイザにロードされる問題は loadbasis でロードしないと効果がありません．maximize または minimize を使って実行した場合には，明示的に loadprob を使ってロードする必要があります．

関連するトピック delbasis，savebasis，XPRSloadbasis (オプティマイザリファレンスマニュアル参照)

loadcuts

目的 オプティマイザにおいて，カットプールから問題にカットの集合をロードする．

概略
```
procedure loadcuts(cuttype: integer, interpret: integer,
    cuts: set of integer)
procedure loadcuts(cuttype: integer, interpret: integer)
```

引数

cuttype		カットを識別するための整数．
interpret		カットの型がインタープリットされる方法．
	-1	全てのカットをロード．
	1	カットの型を数として扱う．
	2	カットの型をビットマップとして扱う．(cuttype のビットセットのどれかにマッチするカットをロード．)
	3	カットの型をビットマップとして扱う．(cuttype のビットセットの全てにマッチするカットをロード．)
cuts		カットプールにあるカットの添字集合．指定されない場合，型 cuttype の全てのカットがロードされる．

モジュール mmxprs

関連する制御 *Boolean*

VERBOSE オプティマイザによって表示されるメッセージを有効/無効にする．

補足 この手続きは，オプティマイザにカットの集合をロードします．カットは全てのノードに継承されます．

関連するトピック addcut，addcuts，delcuts，dropcuts，storecut，storecuts，XPRSloadcuts (オプティマイザリファレンスマニュアル参照)

loadprob

目的 問題をオプティマイザにロードする.

概略
```
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)
```

引数

obj	目的関数制約式
force	必要がなくても行列をロード.
extravar	包含する変数.

モジュール mmxprs

関連する制御 *Boolean*

LOADNAMES オプティマイザへの MPS 名のロードを有効/無効にする.
VERBOSE オプティマイザによって表示されるメッセージを有効/無効にする.

例 以下では，リファレンス番号を 2 にして基底を保存し，問題を変更してオプティマイザにロードし，リファレンス番号 2 の基底をロードしている.

```
declarations
  MinCost: linctr
end-declarations
    ...
savebasis(2)
    ...
loadprob(MinCost)
loasbasis(2)
```

補足 この手続きは，オプティマイザに問題を明示的にロードします．これは，オプティマイザが最後に呼ばれてから問題が変わった場合には，最適化の手続き minimize，maximize によって，自動的に呼ばれます．いくつかの場合には，基底をロードする前に，この手続きを使って明示的に問題をリロード必要があります．最後に loadprob が呼ばれた後に問題が変更された場合には，問題はオプティマイザにリロードされません．引数 force はこのような場合にオプティマイザに問題をロードさせます. 引数 extravar は問題に包含される変数の集合で，制約式には出てこないもの (行列の空の列も) も含みます.

関連するトピック maximize，minimize，XPRSloadlp (オプティマイザリファレンスマニュアル参照)

log

目的 底が 10 の対数の値を返す.

概略 `function log(r: real): real`

引数

`r`	関数に与える実数値．正ではければいけない.

関連する制御 なし.

例 この例は，いろいろな正の底の対数を計算している.

```
function logn(base,number: real): real
  if(number > 0 and base > 0) then
    returned := log(number)/log(base)
  else
    exit(1)
  end-if
end-function
```

関連するトピック `exp,ln.sqrt`

mkdir

目的 ファイルシステムに新しいディレクトリを作る.

概略 `procedure makedir(dirname: string)`

引数

`dirname`	新しく作るディレクトリの名前とパス.

モジュール `mmsystem`

関連する制御 なし.

例 以下では，ログファイルを格納するためのディレクトリを作り，新しいファイルを出力を書き込むために開いている.

```
makedir("logfiles")
if(getsysstat = 0) then
  fopen("logfiles/log1.txt",F_OUTPUT)
    ...
  fclose(F_OUTPUT)
end-if
```

関連するトピック `getsysstat, removedir, system`

makesos

目的 決定変数の集合と線形制約式を使って特殊順序集合 (SOS) を作る.

概略
```
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)
```

引数

cs	線形制約式.
s	決定変数の集合.
c	線形制約式.

関連する制御 なし.

補足 この手続きは，SOS 集合を決定変数の集合 s，線形制約式 c の係数から生成します．結果は cs に返されます.

関連するトピック なし.

maximize

目的 現在の問題を最大化する.

概略 `procedure maximize(alg: integer, obj: linctr)`
`procedure maximize(obj: linctr)`

引数

alg		アルゴリズムの選択. 以下のうちのひとつである.
	XPRS_BAR	ニュートンバリア法.
	XPRS_DUAL	双対単体法.
	XPRS_LIN	大域エンティティを無視して LP を解く.
	XPRS_TOP	LP のあとに停止.
	XPRS_PRI	単体法.
	XPRS_GLB	大域的探索のみ.
	XPRS_NIG	大域的探索の後に iniglobal を呼ばない.
obj		目的関数制約式.

モジュール mmxprs

関連する制御 *Boolean*
LOADNAMES オプティマイザへの MPS 名のロードを有効/無効にする.
VERBOSE オプティマイザによって表示されるメッセージを有効/無効にする.

例 以下では, 目的関数 Profit を双対単体法で最大化し, 大域的探索をする前に停止している.

```
declarations
  Profit: linctr
end-declarations

maximize(XPRS_DUAL+XPRS_TOP,Profit)
```

補足
1. オプティマイザは, 与えられた制約を用いて現在の問題（隠された制約を除いて）を最大化するためにこの手続きを呼びます. 使うアルゴリズムを指定することもできます. デフォルトでは, 問題がグローバルなエンティティを含んでいる場合には自動的に大域的探索を行います. アルゴリズムとパラメータを (プラス記号で) 組み合わせて指定することもできます.
2. XPRS_LIN が定められている場合, 事前解法の手続きの中では, グローバルなエンティティの離散性は, 無視されます.
3. XPRS_TOP が定められている場合, LP をトップのノードで解いて, 分枝限定探索は初期化されません. しかし, グローバルなエンティティ is の離散性は, LP での事前解法でも考慮されます.

関連するトピック initglobal, loadprob, minimize, XPRSmaxim (MAXIM) (オプティマイザリファレンスマニュアル参照)

maxlist

目的 整数と実数のリストの中の最大値を返す．返される型は入力の型に対応する．

概略
```
function maxlist(i1: integer, i2: integer [, i3:
    integer…]): integer
function maxlist(r1: real, r2: real [, r3: real…]): real
```

引数

i1,i2,..	整数のリスト．
r1,r2,..	実数のリスト．

関連する制御 なし．

例 以下では，maxlist によって r の値は 7.0 になる．

```
r := maxlist(-1, 4.5, 2, 7, -0.3)
```

関連するトピック minlist

minimize

目的 現在の問題を最小化する.

概略 `procedure minimize(alg: integer, obj: linctr)`
`procedure minimize(obj: linctr)`

引数

alg		アルゴリズムの選択. 以下のうちのひとつである.
	XPRS_BAR	ニュートンバリア法.
	XPRS_DUAL	双対単体法.
	XPRS_LIN	大域エンティティを無視して LP を解く.
	XPRS_TOP	LP のあとに停止.
	XPRS_PRI	単体法.
	XPRS_GLB	大域的探索のみ.
	XPRS_NIG	大域的探索の後に iniglobal を呼ばない.
obj		目的関数制約式.

モジュール mmxprs

関連する制御 *Boolean*
LOADNAMES オプティマイザへの MPS 名のロードを有効/無効にする.
VERBOSE オプティマイザによって表示されるメッセージを有効/無効にする.

例 以下では, 関数 MinCost をニュートンバリア法で, グローバルなエンティティを無視して最小化している.

```
declarations
  MinCost: linctr
end-declarations

minimize(XPRS_BAR+XPRS_LIN, MinCost)
```

補足

1. オプティマイザは, 与えられた制約をつかって現在の問題（隠された制約を除いて）を最小化するためにこの手続きを呼びます. 使うアルゴリズムを指定することもできます. デフォルトでは, 問題がグローバルなエンティティを含んでいる場合には自動的に大域的探索を行います. アルゴリズムとパラメータを (プラス記号で) 組み合わせて指定することもできます.
2. XPRS_LIN が定められている場合, 事前解法の手続きの中では, グローバルなエンティティの離散性は, 無視されます.
3. XPRS_TOP が定められている場合, LP をトップのノードで解いて, 分枝限定探索は初期化されません. しかし, グローバルなエンティティis の離散性は, LP を用いた事前解法でも考慮されます.

関連するトピック `initglobal`, `loadprob`, `maximize`, `XPRSminim (MINIM)`(オプティマイザリファレンスマニュアル参照)

minlist

目的 整数と実数のリストの中の最小値を返す．返される型は入力の型に対応する．

概略
```
function minlist(i1: integer, i2: integer
    [,i3: integer…]): integer
function minlist(r1: real, r2: real [,r3: real…]): real
```

引数

i1,i2,..	整数のリスト．
r1,r2,..	実数のリスト．

関連する制御 なし．

例 以下では，maxlist によって r の値は-1.0 になる．

```
r := minlist(-1, 4.5, 2, 7, -0.3)
```

関連するトピック maxlist

random

目的 範囲 $[0,1)$ の乱数を発生させて返す.

概略 `function random: real`

引数 なし.

関連する制御 なし.

例 以下では，`i` に 1 と 10 の間の整数の乱数が代入される.

```
setrandseed(round(gettime))
i := integer(round((10*random)+0.5))
```

関連するトピック `setrandseed`

read,readlin

目的 アクティブな入力ストリームからフォーマットされたデータを読む.

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

引数

e1,e2,...	式，式のリスト，または基本的な型.

関連する制御 *Boolean*

IOCTRL インタープリタが IO エラーを無視する/しない.

例 以下では,（数行に分けられているかもしれない）12 45 word を読んだあと，toto(12 and 45)=word を実行している.

```
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))
```

補足

1. これらの手続きは，入力ストリームから与えられた記号にデータを代入するか，入力ストリームから読まれたものと与えられた式を一致させようとします．記号 *ei* が値を代入できる場合，入力ストリームから要求された型の定数を読み込もうとし，成功したら *ei* に値を代入します．この手続きによって間違いなく属性 NBREAD に実際認識された要素の数がセットされます.
2. 手続き read は Mosel の単語解析子に基づいて実行されます．スペースで分けられた要素とスペースを含む文字列は，単引用符または複引用符で囲まれなければなりません.（文字列が識別されたら，引用符は除かれます.）
3. 手続き readlin は一行に含まれる全ての要素が読まれます．readln は行が変わるのを無視します．readln はパラメータを使うことなく行の終わりをスキップします.

関連するトピック NBREAD, SQLreadinteger, SQLreadreal, SQLreadstring, write, writeln

removedir

目的 ディレクトリを削除する.

概略 `procedure removedir(dirname: string)`

引数

`dirname`	削除するディレクトリの名前とパス.

モジュール `mmsystem`

関連する制御 なし.

例 この例では，リスト `dirlist` にある全てのディレクトリを削除する.

```
declarations
  dirlist: set of string
end-declarations
   ...
forall(dir in dirlist)
  if(bittest(getfstat(dir),SYS_TYP) = SYS_DIR) then
   removedir(dir)
  end-if
```

補足 ディレクトリの削除に成功するには，ディレクトリは空でなくてはいけません.

関連するトピック `getfstat`, `getsysstat`, `makedir`

round

目的 数を一番近い整数値に丸める.

概略 `function round(r: real): integer`

引数

`r`	丸める実数値.

関連する制御 なし.

例 以下では，`c` の値は`-2`，`d` の値は 2 になる.

```
c := round(-1.9)
d := round(1.5)
```

関連するトピック `ceil,floor`

savebasis

目的 現在の基底を保存する.

概略 `procedure savebasis(num: integer)`
`procedure savebasis(name: string)`

引数

num	与えられた基底のリファレンス番号.
name	与えられた基底の名前.

モジュール mmxprs

関連する制御 *Boolean*
VERBOSE オプティマイザによって表示されるメッセージを有効/無効にする.

例 以下では，リファレンス番号を 2 にして基底を保存し，ほかの基底をオプティマイザにロードし，古い基底をリロードする.

```
declarations
  MinCost: linctr
end-declarations

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

補足 この関数は，リファレンス番号か名前で与えられる基底の現在の値を保存します. delbasis と loadbasis と一緒に使われます.

関連するトピック delbasis，loadbasis，XPRSwritebasis (オプティマイザリファレンスマニュアル参照)

setcallback

目的 オプティマイザにコールバック関数と手続きをセットする.

概略 `procedure setcallback(cbtype: integer, cb: string)`

引数

`cbtype`	コールバックの型. 可能な値は以下の通りである. `XPRS_CB_IL` 単体法ログコールバック. `XPRS_CB_GL` 大域ログコールバック. `XPRS_CB_BL` バリアログコールバック. `XPRS_CB_USN` ユーザノード選択コールバック. `XPRS_CB_UPN` ユーザプリプロセスノードコールバック. `XPRS_CB_UON` ユーザ最適ノードコールバック. `XPRS_CB_UIN` ユーザ実行不可能ノードコールバック. `XPRS_CB_UIS` ユーザ整数解コールバック. `XPRS_CB_UCN` ユーザカットオフノードコールバック. `XPRS_CB_CMI` カットマネージャ初期化コールバック. `XPRS_CB_CMS` カットマネージャ終了コールバック. `XPRS_CB_CM` カットマネージャ(分枝限定ノード) コールバック. `XPRS_CB_TCM` トップノードカットコールバック.
`cb`	コールバック関数/手続きの名前. パラメータと返り値の型はコールバックによって変化する. `function cb:boolean` .. XPRS_CB_IL, XPRS_CB_GL, XPRS_CB_BL, XPRS_CB_UON, XPRS_CB_CM, XPRS_CB_TCM `function cb(node:integer):integer`..XPRS_CB_USN, XPRS_CB_UPN `procedure cb`..XPRS_CB_UIN, XPRS_CB_UIS, XPRS_CB_CMI, XPRS_CB_CMS `procedure cb(node:integer)`..XPRS_CB_UCN

モジュール `mmxprs`

関連する制御 *Boolean*

`VERBOSE` オプティマイザによって表示されるメッセージを有効/無効にする.

例 以下では, 解の表示を扱う手続きを定義し, 整数解が見つかるといつも呼ばれるようにセットしている.

```
procedure printsol
  declarations
    objval: real
  end-declarations
```

```
  objval := getparam("XPRS_LPOBJVAL")
  writeln("Solution value: ",objval)
end-procedure

setcallback(XPRS_CB_UIS,"printsol")
```

補足 この手続きで，オプティマイザコールバック関数と手続きをセットします．これらのコールバックの詳細については，Xpress-Optimizer ドキュメントを参照してください．解の値は Mosel からコールバック関数/手続きでアクセス可能であること，問題の状態などの他の情報はオプティマイザから関数 getparam で得られることに注意してください

関連するトピック オプティマイザリファレンスマニュアル 5.3 節「コールバックの使用」参照.

setcoeff

目的 変数の係数または定数項をセットする.

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

引数

`c`	線形制約式.
`x`	決定変数.
`r`	係数または定数項.

関連する制御 なし.

例 以下では，制約式 `c` を宣言し，いくつかの項を変更している.

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

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

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

式は，`4*x + 2*y - z ＜＝ -8.1.` に変わっている.

補足 この手続きは，変数が与えられた場合は係数を変え，そうでなければ定数項を変えます.

関連するトピック `getcoeff`，`XPRSchgcoef` (オプティマイザリファレンスマニュアル参照)，`XPRSchgmcoef` (オプティマイザリファレンスマニュアル参照)

sethldden

目的 制約条件を隠したり隠すのをやめたりする.

概略 `procedure sethidden(c: linctr, b: boolean)`

引数

`c`	線形制約式.
`b`	制約の状態.
	`true` 制約を隠す.
	`false` 制約を隠すのをやめる.

関連する制御 なし.

例 以下では，制約式を定義して，それを隠す.

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

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

補足 制約が作られると現在の問題に加えられますが，この手続きを用いて隠すことも可能です．これは，オプティマイザによって問題が解かれるときに，制約条件を含めないが，Mosel の問題の定義から削除されるわけではないことを意味します．関数 `ishiddden` によって，制約の現在の状態をテストすることができます.

関連するトピック `ishidden`

setlb

目的 変数の最小値をセットする.

概略 `procedure setlb(x: mpvar, r: real)`

引数

`x`	決定変数.
`r`	最小値.

モジュール `mmxprs`

関連する制御 *Boolean*

`VERBOSE` オプティマイザによって表示されるメッセージを有効/無効にする.

例 以下では，オプティマイザに問題がロードされ，変数 `b` の最小値が 50 にセットされて `profit` が最大化される.

```
loadprob(profit)
setlb(b,50)
maximize(profit)
```

補足 この手続きによって，直接オプティマイザの中の変数の最小値を変えます．つまり，Mosel が保持している問題の定義には記録では変更されません．直ちに変更されるので，オプティマイザの中で問題をリロードする必要はありません.

関連するトピック `getlb`，`getub`，`loadprob`，`setub`，`XPRSchgbounds` (オプティマイザリファレンスマニュアル参照)

setmlpdlr

目的 変数または特殊順序集合に対する命令をセットする.

概略
```
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)
```

引数

x	決定変数.
c	線形制約式.
r	実数値.
t	命令の型. 以下のうちのひとつである.
	XPRS_PR r は優先順位（1 から 1000 の値をとり，1 が一番優先順位が高い.）
	XPRS_UP 上から順.
	XPRS_DN 下から順.
	XPRS_PU r は擬似コストの上から.
	XPRS_PO r は擬似コストの下から.

モジュール mmxprs

関連する制御 *Boolean*

VERBOSE オプティマイザによって表示されるメッセージを有効/無効にする.

補足 この手続きは，グローバルなエンティティへの命令をセットします. 優先順位の値は整数に変換されることに注意してください. 命令は，問題と同時にオプティマイザにロードされます.

関連するトピック clearmipdir，XPRSgetdirs (オプティマイザリファレンスマニュアル参照) ,XPRSloaddirs (オプティマイザリファレンスマニュアル参照)

setmodcut

目的 制約をモデルのカットとしてマークする.

概略 `procedure setmodcut(c: linctr)`

引数

`c` 線形制約式.

モジュール `mmxprs`

関連する制御 *Boolean*

`VERBOSE` オプティマイザによって表示されるメッセージを有効/無効にする.

例 以下では，問題を解く前に，全ての制約式をモデルのカットとしてセットしている．最後に，モデルのカットが削除されている.

```
declarations
  Items: set of integer
  ctr: array(Items) of linctr
end-declarations
    ...
forall(i in Items|isodd(i))
  setmodcut(ctr(i))
maximize(profit)
writeln("Profit: ",getobjval)
clearmodcut
```

補足 この手続きは，与えられた制約をモデルのカットにします．モデルのカットのリストは行列がロードされたときにオプティマイザに送られます.

関連するトピック `clearmodcut`，`XPRSloadmodelcuts` (オプティマイザリファレンスマニュアル参照)

setparam

目的 制御パラメータの値をセットする.

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

引数

name	値をセットする制御パラメータ（大文字小文字を区別する）. 制御パラメータのリストは 6 章「制御パラメータ」を参照.
val	制御パラメータに与える新しい値.

関連する制御 6 章「制御パラメータ」参照.

例 以下によって，オプティマイザによってメッセージが表示されるようになる.

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

補足 制御パラメータは，Mosel だけでなくロードされているモジュールの設定を含みます. パラメータの名前にモジュールの名前が前置されているので (例えば mmxprs.XPRS_LOADNAMES)，モジュールの名前が分かります. 値の型はパラメータの型に対応していなければなりません.

関連するトピック getparam, オプティマイザリファレンスマニュアルの 7 章「制御パラメータ」参照.

setrandseed

目的 乱数のジェネレータを初期化する.

概略 `procedure setrandseed(s: integer)`

引数

`s`	シードの値.

関連する制御 なし.

例 以下では，`i` に 1 と 10 の間の乱数が代入される.

```
setrandseed(round(gettime))
i := integer(round((10*random)+0.5))
```

補足 この手続きは，引数を，関数 `random` が生成する新しい擬似乱数の系列のシードになります.

関連するトピック `random`

settype

目的 制約式の型をセットする.

概略 `procedure settype(c: linctr, type: integer)`

引数

`c`	線形制約式.
`type`	制約式の型.

関連する制御 なし.

例 この例は，目的関数の最適化の前に制約式の型を `ctr` に変える.

```
declarations
  ctr: linctr
end-declarations
   ...
settype(ctr,CT_EQ)
maximize(profit)
```

補足 線形制約式の型は以下のうちのひとつです.

`CT_EQ`	等しい「=」.
`CT_GEQ`	同じかそれより大きい「≥」.
`CT_LEQ`	同じかそれより小さい「≤」.
`CT_UNB`	バインドされていない制約式.
`CT_SOS1`	型 1 の特殊順序集合.
`CT_SOS2`	型 2 の特殊順序集合.

単項制約式の値は以下の通りです.

`CT_CONT`	連続している.
`CT_INT`	整数である.
`CT_BIN`	バイナリである.
`CT_PINT`	整数の一部である.
`CT_SEC`	準連続である.
`CT_SINT`	準連続な整数である.

関連するトピック `gettype`, `XPRSchgrowtype` (オプティマイザリファレンスマニュアル参照)

setub

目的 変数の最大値をセットする.

概略 `procedure setub(x: mpvar, r: real)`

引数

`x`	決定変数.
`r`	最大値.

モジュール `mmxprs`

関連する制御 *Boolean*

`VERBOSE` オプティマイザによって表示されるメッセージを有効/無効にする.

例 以下では，オプティマイザに問題がロードされ，変数の最大値がセットされて問題が解かれている.

```
loadprob(profit)
setub(a,100)
maximize(profit)
```

補足 この手続きによって，直接オプティマイザの中の変数の最大値を変えます．つまり，Mosel が保持している問題の定義には記録では変更されません．直ちに変更されるので，オプティマイザの中で問題をリロードする必要はありません．よって，最大値や最小値をセットする前には `loadprob` によって明示的に問題をロードする必要があります.

関連するトピック `getlb`，`getub`，`loadprob`，`setlb`，`XPRSchgbounds` (オプティマイザリファレンスマニュアル参照)

sin

目的 値のサイン (正弦) を返す.

概略 `function sin(r:real): real`

引数

`r`	三角関数に与える実数値.

関連する制御 なし.

例 以下では，モデルは IVE の関数を使って正弦関数のグラフを書く.

```
model sine
  uses "mmive";
  IVEinitgraph("Oscillator","x","sin x")
  forall(i in 1..400)
    IVEaddtograph(i,sin(i*0.1))
end-model
```

関連するトピック `srctan,cos`

SQLconnect

目的 接続の文字列で定義されたデータベースとの接続を確立する.

概略 `procedure SQLconnect(s: string)`

引数

`s` 接続の文字列.

モジュール `mmodbs`

関連する制御 *Boolean*

`SQLVERBOSE` ODBC ドライバによって表示されるメッセージを有効/無効にする.

例 以下では,「MySQL」データベース「test」に，ユーザ「yve」，パスワード「DaSH」で接続する.

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

補足 いくつかの接続を開くことは可能ですが，最後に開いた接続がアクティブになります．各接続には識別番号が割り当てられます．その値は，この手続きのあとでは，制御 `SQLCONNECTION` の値によって得られます．この制御は，アクティブな接続を変えることにも使います.

関連するトピック `SQLdisconnect`，`SQLexecute`，`SQLreadinteger`，`SQLreadreal`，`SQLreadstring`

SQLdisconnect

目的 データベースとの接続を切る.

概略 `procedure SQLdisconnect`

引数 なし.

モジュール `mmodbs`

関連する制御 *Boolean*

`SQLVERBOSE` ODBC ドライバによって表示されるメッセージを有効/無効にする.

例 以下では，データベース「test」にユーザ「yve」が接続して，整数値を得て，最後に接続を切っている.

```
SQLconnect("DSN=mysql;DB=test;UID=yves;PWD=DaSH")
i := SQLreadinteger("select articlenum from pricelist
        where colour=blue")
SQLdisconnect
```

補足 アクティブな接続は制御 `SQLCONNECTION` をセットすることによって変えられます.

関連するトピック `SQLconnect`, `SQLexecute`, `SQLreadinteger`, `SQLreadreal`, `SQLreadstring`

SQLexecute

目的 与えられた SQL コマンドを実行する.

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

引数

`s`	実行する SQL コマンド.
`a`	基本的な型（integer,real,string,boolean）の配列.
`m`	基本的な型（integer,real,string,boolean）の集合.

モジュール `mmodbs`

関連する制御

Boolean

`SQLNDXCOL` 最初の列を添字であると解釈する/しない.

`SQLVERBOSE` ODBC ドライバによって表示されるメッセージを有効/無効にする.

Integer

`SQLCOLSIZE` 交換するデータの文字列の最大長. `SQLCONNECTION` アクティブな ODBC 接続の識別番号.

例 以下の例は，4 つの `SQLexecute` 文を含み，以下の作業を行う.

- 列 `colour` の全ての異なる値をテーブル `priicelist` に入れる.
- 配列 `colour`，`prices` を，テーブル `priicelist` の列 `colour`，`prices` の値で初期化する.
- アクティブなデータベースに 2 つの列 `ndx`，`prices` をもつ新しいテーブルを作る.
- テーブル `newtab` にデータのエントリを加える.

```
declarations
  prices: array(1001..1004) of real
  colours: array(1001..1004) of string
  allcolours: set of string
end-declarations
    ...
SQLexecute("select colour from pricelist",allcolours)
SQLexecute("select articlenum,colour,price from
      pricelist",[colours,prices])
SQLexecute("create table newtab (ndx integer, price
      double)")
SQLexecute("insert into table newtab (ndx, price) value
      (?,?)", prices)
```

補足 接続とそのサポートに関するより多くの情報は，データベースドライバのドキュメントを参照してください．

関連するトピック SQLconnect，SQLdisconnect，SQLreadinteger，SQLreadreal，SQLreadstring，SQLSUCCESS

SQLreadinteger

目的 アクティブなデータベースのデータテーブルから整数値を返す.

概略 `function SQLreadinteger(s: string): integer`

引数

s	読む値を選ぶための SQL コマンド.

モジュール `mmodbs`

関連する制御

Boolean

SQLNDXCOL 最初の列を添字であると解釈する/しない.

SQLVERBOSE ODBC ドライバによって表示されるメッセージを有効/無効にする.

Integer

SQLCOLSIZE 交換するデータの文字列の最大長. SQLCONNECTION アクティブな ODBC 接続の識別番号.

例 以下では，フィールド colour を blue であるようなテーブル pricelist の最初のデータのアーティクル番号を読んでいる.

```
i := SQLreadinteger("select articlenum from pricelist
    where colour=blue")
```

補足

1. 整数が見つからなかった場合には 0 を返します.
2. SQL 選択コマンドがひとつの値を表していない場合，条件を満たす最初の値が返されます.

関連するトピック read，readln，SQLexecute，SQLreadreal，SQLreadstring

SQLreadreal

目的 アクティブなデータベースのデータテーブルから実数値を返す.

概略 `function SQLreadreal(s: string): real`

引数

`s`	読む値を選ぶための SQL コマンド.

モジュール `mmodbs`

関連する制御

Boolean

`SQLNDXCOL` 最初の列を添字であると解釈する/しない.

`SQLVERBOSE` ODBC ドライバによって表示されるメッセージを有効/無効にする.

Integer

`SQLCOLSIZE` 交換するデータの文字列の最大長. `SQLCONNECTION` アクティブな ODBC 接続の識別番号.

例 以下では，テーブル `newtab` の添字 2 の `price` のデータが返される.

```
r := SQLreadreal("select price from newtab where ndx=2")
```

補足

1. 実数が見つからなかった場合には 0 を返します.
2. SQL 選択コマンドがひとつの値を表していない場合，条件を満たす最初の値が返されます.

関連するトピック `read`, `readln`, `SQLexecute`, `SQLreadinteger`, `SQLreadstring`

SQLreadstring

目的 アクティブなデータベースのデータテーブルから文字列を返す.

概略 `function SQLreadstring(s: string): string`

引数

s	読む値を選ぶための SQL コマンド.

モジュール `mmodbs`

関連する制御

Boolean

SQLNDXCOL 最初の列を添字であると解釈する/しない.

SQLVERBOSE ODBC ドライバによって表示されるメッセージを有効/無効にする.

Integer

SQLCOLSIZE 交換するデータの文字列の最大長. SQLCONNECTION アクティブな ODBC 接続の識別番号.

例 テーブル pricelist のアーティクル番号 1004 のデータ colour を読んでいる.

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

補足

1. 文字列が見つからなかった場合には空の文字列を返します.
2. SQL 選択コマンドがひとつの値を表していない場合，条件を満たす最初の文字列が返されます.

関連するトピック read，readln，SQLexecute，SQLreadinteger，SQLreadreal

sqrt

目的 値の正の平方根を返す.

概略 `function sqrt(r: real): real`

引数

`r`	関数に与える実数値．非負でなければいけない.

関連する制御 なし.

例 以下は，整数が素数であるかどうかを判定する関数である.

```
function isPrime(number: integer): boolean
  if(number < 2) then returned := false
  end-if

  i := 1
  repeat i+=1
  until((number mod i = 0) or i > sqrt(number))

  if(i > sqrt(number)) then returned := true
  else returned := false
  end-if
end-function
```

関連するトピック `exp,ln.log`

storecut

目的 カットプールにカットを格納し，その添字を返す.

概略 `function storecut(nodupl: integer, cuttype: integer,`
`type: integer, linexp: linctr): integer`

引数

`nodupl`	重複したエントリの扱いを示すフラグ.
	0 チェックなし.
	1 同じカットの型の中で重複をチェック.
	2 全てのカットとの間で重複をチェック.
`cuttype`	カットを識別するための整数の配列.
`type`	カットの型の配列．以下のうちのひとつ.
	`CT_GEQ` 不均等（同じかそれより大きい）
	`CT_LEQ` 不均等（同じかそれより小さい）
	`CT_EQ` 均等
`linexp`	線形式の配列.（値をとる範囲が決まっていない制約式）

モジュール `mmxprs`

関連する制御 *Boolean*

`VERBOSE` オプティマイザによって表示されるメッセージを有効/無効にする.

補足 この関数は，カットを問題の現在のノードに適用しないでカットプールに格納します．カットは `loadcuts` を使ってロードすると現在のノードでアクティブになります.

関連するトピック `addcut`，`addcuts`，`dropcuts`，`delcuts`，`loadcuts`，`storecuts`，`XPRSstorecuts` (オプティマイザリファレンスマニュアル参照)

storecuts

目的 カットプールにカットの配列を格納する.

概略
```
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)
```

引数

nodupl	重複したエントリの扱いを示すフラグ.
	0 チェックなし.
	1 同じカットの型の中で重複をチェック.
	2 全てのカットとの間で重複をチェック.
cuttype	カットを識別するための整数の配列.
type	カットの型の配列. 以下のうちのひとつ.
	CT_GEQ 不均等（同じかそれより大きい）
	CT_LEQ 不均等（同じかそれより小さい）
	CT_EQ 均等
linexp	線形式の配列.（値をとる範囲が決まっていない制約式）
ndx_a	格納したカットの添字番号の間隔.
ndx_b	格納したカットの添字集合.

モジュール mmxprs

関連する制御 *Boolean*

VERBOSE オプティマイザによって表示されるメッセージを有効/無効にする.

補足 この関数は，カットの配列を問題の現在のノードに適用しないでカットプールに格納します．カットは loadcuts を使ってロードすると現在のノードでアクティブになります．カットマネージャは，格納されているカットの添字を配列 ndx_a または整数の集合 ndx_b の形で返します．この手続きでは，パラメータとして 4 つの配列が同じ添字集合でなければいけないことに注意してください.

関連するトピック addcut，addcuts，dropcuts，delcuts，loadcuts，storecut，XPRSstorecuts (オプティマイザリファレンスマニュアル参照)

strfmt

目的 文字列か番号から形式の決められた文字列を作る.

概略
```
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
```

引数

str	フォーマットする文字列.
i	フォーマットする整数.
r	フォーマットする実数.
len	確保されている文字列の長さ（与えられた文字列がこれより長ければ，左に寄せられる）. < 0 確保されている文字列で左寄せ. > 0 確保されている文字列で右寄せ. 0　ユーザデフォルト.
dec	十進の小数点以下の最大の長さ.

関連する制御 なし.

例 1 以下では，出力は「text1 text2text3」になる.

```
writeln("text1", strfmt("text2",8), "text3")
```

例 2 以下では，出力は「789.123 789.12　　789」になる.

```
r := 789.123456
writeln(strfmt(r,0)," ", strfmt(r,4,2), strfmt(r,8,0))
```

補足
1. この関数は，文字列か整数か実数からフォーマットされた文字列を作ります．文字列を使う場所ではどこでも使用できますが，フォーマットされた出力をつくるために (write や writeln と一緒に) よく使います.
2. 結果の文字列が確保されているより長かった場合にはカットはされませんが，確保されている場所の右にあふれてプリントされます.

関連するトピック substr, write, writeln

substr

目的 文字列の一部を返す.

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

引数

str	文字列.
i1	部分文字列の始まりの場所.
i2	部分文字列の終わりの場所.

関連する制御 なし.

例 以下では，出力が「ample te」になる.

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

関連するトピック なし.

system

目的 システムコマンドを実行する.

概略 `procedure system(command: string)`

引数

`command`	実行するコマンド.

モジュール `mmsystem`

関連する制御 なし.

例 以下では，新しいディレクトリ `Newdir` を作る.

```
system("mkdir Newdir")
```

補足 この手続きは，異なるシステムで実行されるアプリケーションでは，システムに依存するため使わないほうがよいです.

関連するトピック `getsysstat`

write,writeln

目的 アクティブな出力ストリームに式または式のリストを送る.

概略
```
procedure write(e1: expr [,e2: expr…])
procedure writeln
procedure writeln(e1: expr [,e2: expr…])
```

引数

e1,e2,..	式または式のリスト.

関連する制御 *Boolean*

IOCTRL インタープリタが IO エラーを無視する/しない.

例 以下では, 出力の集合 Set 1 を定義し, 空行をプリントしている. 出力は「A real:7.12, a Boolean:true」になる.

```
Set1 := {"first", "second", "fifth"}
write(Set1)
writeln
b := true
writeln("A real:", strfmt(7.1234, 4, 2), ", a
      Boolean:",b)
```

補足 これらの手続きは, アクティブな出力ストリームに与えられた式を書きます. 手続き writeln は出力の最後にリターン文字を加えます. 数は, 関数 strfmt を使ってフォーマットするべきです. 基本的な型は「そのまま」表示されます. linctr,mpvar 型では, アドレスだけがプリントされます. 式が集合か配列の場合, 全ての要素が表示されます.

関連するトピック fselect, read, readln, strfmt

第 5 章

コンソール関数とライブラリ関数

Xpress-Mosel のコンソールユーザ，および，ライブラリユーザーは，モデルのコンパイル，ロード，実行のための簡単なルーチンから，動的共有オブジェクト (モジュール) の管理のためのルーチンにいたるまで，多数のルーチンが利用可能です．コアルーチンは，このような機能をコンソール，ライブラリの双方のユーザーに提供します．しかし，コンソールユーザには，作業を行うために 1 つの関数で十分であるときにも，対応するいくつかのライブラリ関数存在していることを覚えておきましょう．また，ライブラリは，モデル化のプロセスにとって，更に多くの制御を可能にする追加の関数を含んでいます．

ヘッダファイル　ここで説明するライブラリ関数は，Mosel ランタイムライブラリ (xprm_rt) とモデルコンパイラライブラリ (xprm_mc) の両方ですが，関数の説明で何も言われていなければランタイムライブラリの一部です．(現在は，XPRMcompmod はモデルコンパイラライブラリの一部です)． ライブラリ関数を利用する場合には，対応するヘッダファイルをプログラムの最初に包含しなければなりません．ヘッダファイルは，xprm_rt.h と xprm_mc.h です．

5.1　コンソール関数

コンソールの Mosel のユーザのために，本章では以下のような関数が直接関連します．

コマンド	説明	ページ
CLOAD	モデルファイルをコンパイルして BIM ファイルをロードする.	
COMPILE	モデルファイルをコンパイルして BIM ファイルを生成する.	
DELETE	メモリからモデルを削除する.	
DISPLAY	与えられた記号の値を表示する.	
EXAMINE	与えられたモジュールの情報を提供する.	
EXPRTPROB	行列を表示するかファイルに保存する.	
FLUSHLIBS	使っていない動的共有オブジェクト（DSO）をアンロードする.	
INFO	アクティブなプログラムの情報を表示する.	
LIST	ロードされている全てのモジュールを表示する.	
LOAD	バイナリモデルをロードし，必要なモジュールを開く.	
LSLIBS	ロードされている全ての動的共有オブジェクト（DSO）を表示する.	
QUIT	現在の Mosel セッションを終了する.	
RESRT	アクティブなモデルを再初期化する.	
RUN	アクティブなモデルを実行する.	
SELECT	モデルをアクティブにする.	
SYMBOLS	アクティブなモデルが公表している記号を表示する.	
SYSTEM	OS のコマンドを実行する.	

5.2 関数の説明のためのレイアウト

本章で説明されている関数は，次のような項目に沿っています.

関数の名前

分かりやすくするために，各ルーチンの説明の始まりにページを新しくします．関数のためのライブラリの名前は左に，コンソールの名前は右におきます.

目的

ルーチンの短い説明です．ここから情報を始めるという目的です.

概略

ルーチンの使い方の文法の概略です．ライブラリの文法がはじめに，コンソール Mosel の文法が次に書いてあります.

引数

ルーチンの引数のリストです.

例

1 つか 2 つのルーチン使用例です.

補足

その他の情報です.

関連するトピック

ルーチンに関連するトピックや比較や参照のためのトピックです.

XPRMautounloaddso

目的 動的共有オブジェクトを自動的にアンロードするのを可能または不可能にする.

概略 `void XPRMautounloaddso(int yesno);`

引数

yesno	0 なら不可能，そうでなければ可能.

例 以下の例は，モジュールの自動的なアンロードは無効になっていて，ロードされたたくさんのモデルが実行され，最後に全てのモジュールをアンロードしている.

```
XPRMmodel mod;
int nReturn;
    ...
XPRMautounloaddso(0);
    ...
for(mod = XPRMgetnextmod(NULL); mod != NULL; mod =
        XPRMgetnextmod(mod))
  XPRMrunmod(mod,&nReturn,NULL);
XPRMflushdso();
```

補足 デフォルトでは，使われていないモジュールは一定時間後に自動的にアンロードされます．この関数を使うと自動的なアンロードを無効にできます．使われていないモジュールは，XPRMflushdso を使って明示的にアンロードします.

関連するトピック XPRMflushdso (FLUSHLIBS)

XRPMchkarrind

目的 添字タプルが配列の範囲の中に入っているかどうかをチェックする.

概略 `int XPRMchkarrind(XPRMarray array, int indices[]);`

引数

`array`	配列への参照.
`indices`	配列 `array` の次元が n であるとき，添字の n-タプル.

例 以下の例は，添字 $(4, 4)$ が 2 次元配列 `A` の範囲の中に入っているかどうかを調べている.

```
XPRMmodel model
XPRMalltypes atarr;
XPRMarray A;
int indices[2];
    ...
XPRMfindident(model,"A",&atarr);
A = atarr.array;
indices[0] = 4; indices[1] = 4;
if(XPRMchkarrind(A,indices) != 1)
  printf("Index (%d,%d) lies within the range bounds of
          A\n",indices[0],indices[1]);
```

補足 返り値が 0 であるとき，添字タプルは配列の範囲の中に入っており，1 であるときには入っていません.

関連するトピック `XPRMcmpindices`

CLOAD

目的 モデルのソースファイルをコンパイルしてバイナリモデルファイル (.bim ファイル) にし，それを実行するために Mosel にロードする．

概略 CLOAD *[-options] srcfile [userc]*

引数

options		コンパイルオプション．
	g	デバッグ情報を包含する．モデルを実行中にエラーが起こったときに，ソースファイルの中のどこでエラーが起こったかが示される．
	s	ストリップ記号．BIM ファイルから，ソースモデルで使われたプライベートな記号の名前を全て除き，安全にする．
	m	モデラーエミュレータの文法を使い，.mod というファイル名の拡張子を仮定する．
	M	ファイル名の拡張子が.mod であっても，モデラーエミュレータの文法は使わない．
	p	パースのみ．ソースの構文解析が終わったら，コンパイルはしないで停止する．(ファイルを生成しない.)
	e	自動的なファイル拡張子の付加を無効にする．ファイル名に拡張子を付加しない．
srcfile		ソースファイルの名前．拡張子が付いておらず，オプション e が使われていない場合には，.mos という拡張子を付加する．
userc		出力ファイルの最初にコメントテキストとして保存される．

例 以下は，コンソールからバッチモードで Mosel モデル myprob.mos をコンパイル，ロード，実行している．

```
%> mosel -c 'CL myprob; RUN '
```

補足 CLOAD は，コンソールから COMPILE して直ちに LOAD を呼び出しているのと等価です．

関連するトピック XPRMcompmod (COMPILE), XPRMloadmod (LOAD), XPRMrunmod (RUN)

XPRMcmpindices

目的 2 つの添字タプルを比較する.

概略 `int XPRMcmpindices(int transpose, int nbdim, int ind1[],int ind2[])`

引数

`transpose`	0 でない場合は転置された配列を比較し，そうでなければ普通の順番で比較する.
`nbdim`	次元の数.（タプルのサイズ `ind1`，`ind2` と等しい.）
`ind1`	`nbdim` の添字タプルのサイズ.
`ind2`	`nbdim` の添字タプルのサイズ.

例 以下の例では，配列 `A` は最初のエントリの添字とその最初の真のエントリの添字と比較されている.

```
XPRMalltypes atarr;
XPRMarray A;
int *indices, *trueindices;
    ...
findident(model,"A",&atarr);
A = atarr.array;
indices = malloc(XPRMgetarrdim(A)*sizeof(int));
trueindices = malloc(XPRMgetarrdim(A)*sizeof(int));
XPRMfirstarrentry(A,indices);
XPRMfirstarrtruentry(A,trueindices);

if(XPRMcmpindices(0, XPRMgetarrdim(A), trueindices,
      indices) != 0)
  printf("Dynamic array A has not had its first entry
      defined\n");
```

補足 この関数は，2 つの添字タプルを普通の順番と転置の順番で比べます．返り値は以下のうちのひとつです.

-1 タプル `ind1` はタプル `ind2` より前.

0 2 つのタプルは同じ.

1 タプル `ind2` はタプル `ind1` より前.

関連するトピック `XPRMchkarrind`

XPRMcompmod COMPILE

目的 モデルのソースファイルをコンパイルしてバイナリモデルファイル (.bim ファイル) にする．モデルを実行するための関数 XPRMloadmod (LOAD) の入力として必要.

概略 `int XPRMcompmod(const char *options, const char *srcfile, const char *dstfile, const char *userc);`

COMPILE *[-options] srcfile [userc]*

引数

options		コンパイルオプション.
	g	デバッグ情報を包含する．モデルを実行中にエラーが起こったときに，ソースファイルの中のどこでエラーが起こったかが示される.
	s	ストリップ記号．BIM ファイルから，ソースモデルで使われたプライベートな記号の名前を全て除き，安全にする.
	m	モデラーエミュレータの文法を使い，.mod というファイル名の拡張子を仮定する.
	M	ファイル名の拡張子が.mod であっても，モデラーエミュレータの文法は使わない.
	p	パースのみ．ソースの構文解析が終わったら，コンパイルはしないで停止する.（ファイルを生成しない.）
	e	自動的なファイル拡張子の付加を無効にする．ファイル名に拡張子を付加しない.
srcfile		ソースファイルの名前．拡張子が付いておらず，オプション e が使われていない場合には，.mos という拡張子を付加する.
dstfile		出力ファイルの名前．NULL なら出力は srcfile.bim という名前になる.
userc		出力ファイルの最初にコメントテキストとして保存される.

例 1(ライブラリ) 以下は，Mosel モデルをソースファイル myprob.mos からコンパイル，ロード，実行している.

```
XPRMmodel model;
int nReturn;

XPRMinit();
XPRMcompmod("","myprob.mos",NULL,"My Problem");
model = XPRMloadmod("myprob.bim",NULL);
XPRMrunmod(model,&nReturn,NULL);
XPRMfree();
```

例 2(コンソール) 以下は，モデル myprob.mos をコンパイルし，結果を myprob.bim に保存し，実行している.

```
COMPILE myprob
LOAD myprob
RUN
QUIT
```

補足

1. XPRMcompmod は Mosel モデルコンパイラライブラリの一部なので，これを使うためにはヘッダファイル xprm_mc.h を包含する必要があります.
2. 返り値は以下のうちのひとつです.
   0 関数の実行に成功.
   1 パースで失敗.（文法エラーかファイルアクセスエラー.）
   2 コンパイル時にエラー．(セマンティックエラーの検出．)
   3 出力ファイルの書き込みエラー.
3. 引数 srcfile 以外は必要なければ NULL になっています.
4. ソースファイルの名前が拡張子をつけないで与えられ，オプション-e が選ばれていなければ，Mosel は拡張子.mos を付けます．出力ファイルの名前が与えられていない場合，出力ファイルはソースファイルと同じ名前で拡張子は.bim です．空の文字列 ("") の場合，srcfile は標準入力， dstfile は標準出力になります.

関連するトピック CLOAD，XPRMloadmod（LOAD），XPRMrunmod（RUN）

DELETE

目的 メモリからモデルをアンロードする.

概略 DELETE *[number — name]*

引数

number	モデルの番号. モデルがロードされたときに自動的に割り当てられる.
name	モデルの名前. ソースファイルの model 文で与えられたもの.

例 以下は, Mosel モデル simple.mos をコンパイルしてロードし, 同様に alterd.mos をコンパイルしてロードし,「alterd」をアンロードして「simple」をアクティブな問題にしている.

```
CLOAD simple
RUN
CLOAD altered
RUN
DELETE
```

補足

1. BIM ファイルはこのコマンドでは削除できません.
2. モデルの名前や番号が与えられなければ, アクティブなモデルがアンロードされます. 一番最後にロードされたモデルがアクティブなモデルです.
3. モデル番号は LIST コマンドを使って調べることができます.

関連するトピック CLOAD, LIST, XPRMloadmod (LOAD), SELECT, XPRMunloadmod

DISPLAY

目的 与えられた記号の値を表示する.

概略 DISPLAY *symbol*

引数

symbol	値を表示したいものの記号.

例 以下は，Mosel モデル myprob.mos をコンパイル，ロード，実行し，目的関数の値 (profit) を表示しています.

```
CLOAD myprob
RUN
DISPLAY profit
```

補足 モデルを実行する前には，定数にしかアクセスできません．決定変数の場合は解が (デフォルトは 0). 制約式の場合はアクティビティ値が (デフォルトは 0)，表示されます.

関連するトピック getobjval，getsol，SYMBOLS，XPRMgetcsol，XPRMgetobjval，XPRMgetvsol

EXAMINE

目的 与えられたモジュールの定数，手続き/関数，制御/属性，型のリストを表示する．

概略 EXAMINE *[-options] libname*

引数

options		どの情報を表示するかのオプション．
	c	定数．
	s	手続き/関数（サブルーチン）．
	p	制御/属性（パラメータ）．
	t	型．
libname		情報を表示したいモジュールの名前．

例 以下は，モジュール mmodbc の全ての手続き/関数，制御/属性を表示する．

```
EXAMINE mmodbc
```

関連するトピック XPRMgetdsoinfo，XPRMgetdsoparam，XPRMgetnextdsoconst,XPRMgetnextdsoparam，XPRMgetnextdsoproc

XPRMexportprob EXPORTPROB

目的 問題を MPS または LP 形式の行列ファイルとして出力する.

概略 `int XPRMexportprob(XPRMmodel model, const char *options,const char *fname, XPRMlinctr obj);`

`EXPORTPROB` *[-options] [filename [obj]]*

引数

`model`		モデルへのリファレンス.
`options`		出力する形式. 可能な値は以下の通り.
		LP 形式, 最小化 (デフォルト).
	`m`	MPS 形式.
	`p`	最大化 (LP 形式のときのみ有効.).
	`s`	スクランブルされた名前を使う.
`filename`		ファイル名. 必要ない場合は `NULL`.
`obj`		最適化に使う目的. 必要ない場合は `NULL`.

例 1(ライブラリ) 以下は, コンソールに最大化問題を LP 形式でプリントしている.

```
XPRMmodel model;
int nReturn;
   ...
XPRMrunmod(model,&nReturn,NULL);
XPRMexportprob(model,"p",NULL,NULL)
```

例 2(コンソール) 以下は, ロードされたモデルを実行し, 行列を MPS 形式で `myprob.mat` というファイルに出力している.

```
RUN
EXPORTPROB -m myprob.mat
```

補足 ファイル名が `NULL` の場合, 出力はコンソールになります. ファイル名が拡張子なしで与えられた場合, MPS ファイルなら `.mat`, LP 形式なら `.lp` が付加されます. 出力形式のオプションはひとつの文字列として使うこともできます (例えば ”`sp` ”). この関数は, Mosel が「トライアルモード」で実行されているときは無効にされています.

関連するトピック `exportprob`, `XPRBexportprob`（BCL リファレンスマニュアル参照）

XPRMfinddso

目的 モジュール名から DSO ディスクリプタを返す.

概略 `XPRMdsolib XPRMfinddso(const char *libname);`

引数

`libname`	見つけたいモジュールの名前.

例 以下は，モジュール `mmxprs` へのリファレンスが返されて，バージョン番号が表示されている.

```
XPRMdsolib lib;
int libv;
const char *name;
    ...
if((lib = XPRMfinddso("mmxprs")) != NULL)
{
  XPRMgetdsoinfo(lib,&name,NULL,&libv,NULL);
  printf("Got module: %s version %d.%d.%d\n", name,
        libv/1000000, (libv/1000)-1000*(libv/1000000),
        libv - 1000*(libv/1000));
}
```

補足 もしモジュールがロードされていなかったら，`NULL` が返される.

関連するトピック `XPRMgetnextdso`

XPRMfindident

目的 辞書の中の識別子を見つける.

概略 `int XPRMfindident(XPRMmodel model, const char *text,XPRMalltypes *value);`

引数

`model`	現在のモデル.
`text`	識別子.
`value`	辞書のエントリへのポインタが返される.

例 以下は，決定変数 a と b の解の値が返されている.

```
XPRMalltypes atvar;
XPRMmpvar a, b;
in nReturn;
    ...
XPRMrunmod(simple,&nReturn,NULL);
XPRMfindident(simple,"a",&atvar);
a = atvar.mpvar;
XPRMfindident(simple,"b",&atvar);
b = atvar.mpvar;
printf("a = %g, b = %g\n",XPRMgetsol(a),XPRMgetsol(b));
```

補足 1. XPRMfindident は与えられたモデルの与えられた識別子の辞書のエントリを，そのエントリの型と構造とともに返します．識別子が認識されなかった場合は 0 を返します．型と構造はビットエンコードされており，マクロ XPRM_TYP(t) と XPRM_STR(t) を使って抽出できます．可能な構造体は以下の通りです.

XPRM_STR_CONST	オブジェクトは定数.
XPRM_STR_REF	オブジェクトはスカラーへのリファレンス.
XPRM_STR_SET	オブジェクトは集合.
XPRM_STR_ARR	オブジェクトは配列.
XPRM_STR_PROC	オブジェクトは手続きか関数.

構造体によって，可能な型は以下の通りです.

XPRM_TYP_NOT	型なし（手続き）
XPRM_TYP_INT	整数（定数，リファレンス，集合，配列，関数）
XPRM_TYP_REAL	実数（定数，リファレンス，集合，配列，関数）
XPRM_TYP_STRING	文字列（定数，リファレンス，集合，配列，関数）
XPRM_TYP_BOOL	ブーリアン（定数，リファレンス，集合，配列，関数）
XPRM_TYP_MPVAR	決定変数（リファレンス，集合，配列）
XPRM_TYP_LINCTR	線形制約式（リファレンス，集合，配列）
XPRM_TYP_SET	集合（集合，配列）
XPRM_TYP_ARR	配列（集合，配列）

他の値は外部の型（モジュールによって提供される型）として設計されています．関数 XPRMgettypename によって，型の名前を調べることができます（外部の型のエンティティの値をインタープリッとすることはできません）．

2. 共有体 XPRMalltypes は全ての可能な型を集め，XPRMfindident の呼び出しの結果は構造によって以下のようにデコードされます．

value.integer	定数かリファレンス．
value.real	定数かリファレンス．
value.string	定数かリファレンス．
value.boolean	定数かリファレンス．
value.mpvar	リファレンス．
value.linctr	リファレンス．
value.set	集合 (集合の関数の入力として使われる．)
value.array	配列 (配列の関数の入力として使われる．)
value.proc	手続きと関数．

関連するトピック XPRMgetnextident

XPRMfindmod

目的 名前か命令番号からモデルをみつける.

概略 `XPRMmodel XPRMfindmod(const char *name, int number);`

引数

`name`	モデルの名前か NULL.
`number`	モデルの命令番号か -1.

例 以下は，モデルの問題へのポインタが，モデルの中であとでみつけられる様子を示している.

```
XPRMmodel myprob;

XPRMinit();
XPRMloadprob("myprob.bim","My Problem");
   ...
myprob = XPRMfindmod("My Problem",-1);
```

補足

1. XPRMfindmod はモデルへのポインタを返す．モデルが存在していないときは NULL を返します.
2. ロードされているモデルのリストでは，各モデルは内部の名前 (.bim ファイルに格納されている名前であり，ファイル名ではない．) と命令番号（モデルがロードされたときに自動的に割り当てられる.）で特徴付けられています．この関数は，名前 (number $= -1$) と命令番号 (number =NULL) のどちらかで識別されるモデルを返します．両方の引数が定義されている場合，name で定義されるモデルへのポインタを返します.

関連するトピック XPRMloadmod

XPRMfirstarrentry

目的 配列の最初のエントリの添字のリストを得る.

概略 `int XPRMfirstarrentry(XPRMarray array, int indices[]);`

引数

`array`	配列へのリファレンス.
`indices`	配列の最初の論理要素の添字である n-タプル（n は配列 `array` の次元）が返される.

例 以下では，配列 `A` の最初のエントリの添字が返されて表示されている.

```
XPRMmodel model;
XPRMalltypes atarr;
XPRMarray A;
int i,*indices;
    ...
XPRMfindident(model,"A",&atarr);
A = atarr.array;
indices = malloc(XPRMgetarrdim(A)*sizeof(int));
XPRMfirstarrentry(A,indices);
for(i=0;i<XPRMgetarrdim(A);i++)
  printf("Index of first entry of A in dimension %d is
         %d\n",i+1,indices[i]);
```

関連するトピック `XPRMfirstarrtruentry`, `XPRMfirstsetndx`, `XPRMlastarrentry`, `XPRMnextarrentry`

XPRMfirstarrtruentry

目的 配列の最初の真のエントリの添字のリストを返す.

概略 `int XPRMfirstarrtruentry(XPRMarray array, int indices[]);`

引数

`array`	配列へのリファレンス.
`indices`	配列の最初に定義されている要素の添字である n-タプル（n は配列 `array` の次元）が返される.

例 以下では，動的な配列 `A` の最初のエントリの添字が返されて表示されている.

```
XPRMmodel model;
XPRMalltypes atarr;
XPRMarray A;
int i,*indices;
    ...
XPRMfindident(model,"A",&atarr);
A = atarr.array;
indices = malloc(XPRMgetarrdim(A)*sizeof(int));
XPRMfirstarrtruentry(A,indices);
for(i=0;i<XPRMgetarrdim(A);i++)
    printf("Index of first true entry of A in dimension
         %d is %d\n",i+1,indices[i]);
```

補足 固定されたサイズの配列（密な配列）の場合，関数 `XPRMfirstarrentry` のように動作します．動的な配列の場合，配列の最初の真のエントリの添字タプルを返します.

関連するトピック `XPRMfirstarrentry`, `XPRMlastarrentry`, `XPRMnextarrtruentry`

XPRMfirstsetndx

目的 与えられた集合の最初の要素の添字を返す.

概略 `int XPRMfirstsetndx(XPRMset set);`

引数

`set`	集合へのリファレンス.

例 以下では，集合 `Items` の要素が検索されてプリントされている.

```
XPRMmodel model;
XPRMalltypes atset, atelt;
XPRMset items;
    ...
XPRMfindident(model,"Items",&atset);
items = atset.set;
if(XPRM_TYP(XPRMgetsettype(items)) == XPRM_TYP_INT)
  for(i=firstsetndx(items);i<=XPRMlastsetndx(items);i++)
  {
    XPRMgetelsetval(items,i,&atelt);
    printf("Items[%d] = %d\n",j,atelt.integer);
  }
```

補足 1. 範囲集合の場合，最小値が返されます．string の集合の場合，最初の要素は必ず添字 1 になります.

関連するトピック `XPRMgetelsetndx, XPRMgetelsetval, XPRMgetsetsize,XPRMlastsetndx`

XPRMflushdso FLUSHLIBS

目的 使っていない動的共有オブジェクトをアンロードする.

概略 `void XPRMflushdso(void);`
`FLUSHLIBS`

引数 なし.

例 1(ライブラリ) 以下の例では，使っていないモジュールがアンロードされている．その後，モデルの全てのモジュールへのリストがバージョンとともに表示されている.

```
XPRMdsolib lib;
const char *name;
int libv;
    ...
XPRMflushdso();
for(lib=NULL;(lib = XPRMgetnextdso(lib))!= NULL;)
{
  XPRMgetdsoinfo(lib,&name,NULL,&libv,NULL);
  printf("Model uses %s, version %d.%d.%d\n", name,
      libv/1000000, (libv/1000)-1000*(libv/1000000),
      libv-1000*(libv/1000));
}
```

例 2(コンソール) 以下では，ロードされているモデルを実行し，使っていないモジュールをアンロードし，その後，提供されているモジュールが提供されている.

```
RUN
FLUSHLIBS
LSLIBS
```

補足 使っていない各モジュールは，一定時間後に自動的にアンロードされます．この関数は，マネージャに使っていないモジュールをアンロードさせます.

関連するトピック `fflush`, `XPRMautounloaddso`

XPRMfree

目的 Mosel によって使われたモジュールをアンロードし，メモリを開放して，Mosel セッションを終わらせます.

概略 `int XPRMfree(void);`

引数 なし.

例 以下では，モデルをロードして実行し，全てのリソースを開放しています.

```
XPRMmodel model;
int nReturn;

XPRMinit();
model = XPRMloadmod("myprob",NULL);
XPRMrunmod(model,&nReturn,NULL);
XPRMfree()
```

関連するトピック `XPRMinit`, `XPRSfree`（オプティマイザリファレンスマニュアル参照）

XPRMgetact

目的 線形制約式のアクティビティ値を得る.

概略 `double XPRMgetact(XPRMmodel model, XPRMlinctr ctr);`

引数

`model`	現在のモデル.
`ctr`	線形制約式へのリファレンス.

例 この例では，問題を解いた後の制約式 `constraint` のアクティビティ値を表示している.

```
XPRMmodel model;
XPRMalltypes atctr;
XPRMlinctr constraint;
int nReturn;
    ...
XPRMrunmod(model,&nReturn,NULL);
XPRMfindident(model,"constraint",&atctr);
constraint = atctr.linctr;
printf("Activity for constraint: %g\n",
        XPRMgetact(model,constraint);
```

補足 `XPRMgetact` は，問題を解くのに既に成功している場合に与えられた線形制約式のアクティビティ値を返します.

関連するトピック `getact`

XPRMgetarrdim

目的 配列の次元を返す.

概略 `int XPRMgetarrdim(XPRMarray array);`

引数

`array`	配列へのリファレンス.

例 以下では，モデルから配列 `x` を検索し，次元と大きさをプリントしている.

```
XPRMmodel model;
XPRMalltypes atarr;
XPRMarray x;
    ...
XPRMfindident(model,"x",&atarr);
x = atarr.array
printf("dimension of x: %d, size of x: %d\n",
        XPRMgetarrdim(x),XPRMgetarrsize(x));
```

関連するトピック `XPRMchkarrind`, `XPRMgetarrsets`, `XPRMgetarrsize`, `XPRMgetarrtype`

XPRMgetarrsets

目的 配列の添字集合を得る.

概略 `void XPRMgetarrsets(XPRMarray array, XPRMset sets[]);`

引数

`array`	配列へのリファレンス.
`sets`	n 次元配列 `array` について，配列の添字集合の n-タプル.

例 以下では，配列 `x` の添字集合を探し，配列 `sets` に格納している.

```
XPRMmodel model;
XPRMalltypes atarr;
XPRMarray x;
XPRMset *sets;
    ...
XPRMfindident(model,"x",&atarr);
x = atarr.array;
sets = malloc(XPRMgetarrdim(x)*sizeof(XPRMset));
XPRMgetarrsets(x,sets);
```

関連するトピック `XPRMgetarrdim`, `XPRMgetarrsize`, `XPRMgetarrtype`, `XPRMgetsetsize`

XPRMgetarrsize

目的 配列のサイズ（真のエントリの総数）を返す.

概略 `int XPRMgetarrsize(XPRMarray array);`

引数

`array`	配列へのリファレンス.

例 以下では，配列 `x` をモデルから検索し，次元とサイズをプリントしている.

```
XPRMmodel model;
XPRMalltypes atarr;
XPRMarray x;
    ...
XPRMfindident(model,"x",&atarr);
x = atarr.array
printf("dimension of x: %d, size of x: %d\n",
        XPRMgetarrdim(x),XPRMgetarrsize(x));
```

関連するトピック `XPRMgetarrdim`, `XPRMgetarrsets`, `XPRMgetarrtype`

XPRMgetarrtype

目的 配列の型を返す.

概略 `int XPRMgetarrtype(XPRMarray array);`

引数

`array`	配列へのリファレンス.

例 以下では，配列 x をモデルから検索し，型を調べている.

```
XPRMmodel model;
XPRMalltypes atarr;
XPRMarray x;
    ...
XPRMfindident(model,"x",&atarr);
x = atarr.array;
switch(XPRM_TYP(XPRMgetarrtype(x)))
{
  case XPRM_TYP_NOT: printf("no type\n"); break;
  case XPRM_TYP_INT: printf("integer\n"); break;
  case XPRM_TYP_REAL: printf("real\n"); break;
  case XPRM_TYP_STRING: printf("string\n"); break;
  case XPRM_TYP_MPVAR: printf("mpvar\n"); break;
  case XPRM_TYP_LINCTR: printf("linctr\n"); break;
  default: printf("Could not determine type.\n");
}
```

補足 配列の型は全てのエントリと配列の記憶クラスによって決まります. エントリの型は, マクロ XPRM_TYP(type) を使って抽出できます. 記憶クラスは, マクロ XPRM_GRP(type) を使って抽出できます. マクロ XPRM_ARR_DENSE は,「密なテーブル」を特徴付けるために使います (たとえば, XPRM_GRP(type) == XPRM_ARR_DENSE)).

関連するトピック XPRMfindident, XPRMgetarrdim, XPRMgetarrsets, XPRMgetarrsize

XPRMgetarrval

目的 配列の与えられた添字タプルに対応するエントリの値を返す.

概略 `int XPRMgetarrval(XPRMarray array, int indices[],void *adr);`

引数

`array`	配列へのリファレンス.
`indices`	n 次元配列 `array` について，配列の添字集合の n-タプル.
`adr`	添字タプルが返される場所を記述する配列のエントリの値を格納する場所へのポインタ.

例 以下では，動的な配列 `A` の最初の真のエントリの値を返している.

```
XPRMmodel model;
XPRMalltypes atarr;
XPRMarray A;
int *indices, value;
    ...
XPRMfindident(model,"A",&atarr);
A = atarr.array;
indices = malloc(XPRMgetarrdim(A)*sizeof(int));
XPRMfirstarrtruentry(A,indices);
XPRMgetarrval(A,indices,&value);
printf("The first array value is %d\n",value);
```

補足 配列の型のデータを受け取るには，十分大きい領域が確保されている必要があります．例えば，real の配列（型は `XPRM_TYP_REAL`）では，引数 `adr` は型 `double *`である必要があります.

関連するトピック `XPRMchkarrind`, `XPRMfirstarrentry`, `XPRMfirstarrtruentry`, `XPRMgetelsetval`, `XPRMlastarrentry`, `XPRMnextarrentry`

XPRMgetcsol

目的 線形制約式の解の値を返す.

概略 `double XPRMgetcsol(XPRMmodel model, XPRMlinctr ctr);`

引数

`model`	現在のモデル.
`ctr`	線形制約式へのリファレンス.

例 以下では，制約式の解の値の見つけ方を示す.

```
XPRMmodel model;
XPRMalltypes atctr;
XPRMlinctr first;
int nReturn;
    ...
XPRMrunmod(model,&nReturn,NULL);
XPRMfindident(model,"first",atctr);
first = ctr.linctr;
printf("%g\n",XPRMgetcsol(model,first));
```

補足 `XPRMgetcsol` は現在の解を使って与えられた制約式を評価して値を返します（Mosel の関数 `getsol` に対応します.）.

関連するトピック `getsol`, `XPRMgetdual`, `XPRMgetrcost`, `XPRMgetslack`, `XPRMgetvsol`

XPRMgetctrnum

目的 線形制約式の行の数を返す.

概略 `int XPRMgetctrnum(XPRMlinctr ctr);`

引数

`ctr`	線形制約式へのリファレンス.

例 以下では，制約式 `first` と `second` の行の数を調べて表示している.

```
XPRMmodel mod;
XPRMalltypes atvar;
XPRMlinctr first, second;
    ...
XPRMfindident(mod,"first",&atvar);
first = atvar.linctr;
XPRMfindident(mod,"second",&atvar);
second = atvar.linctr;

printf("constraint  'first ' has number %d\n
        constraint  'second ' has number %d\n",
        XPRMgetctrnum(first), XPRMgetctrnum(second));
```

補足 問題が得られないときやメインの問題に制約式がない場合には，負の値を返します.

関連するトピック `XPRMgetvarnum`，`XPRSgetrows`（オプティマイザリファレンスマニュアル参照)

XPRMgetdsoinfo

目的 動的共有オブジェクトに関する情報を得る.

概略 `void XPRMgetdsoinfo(XPRMdsolib dso, const char **name,int *nb, int *version, int *nbref);`

引数

`dso`	Mosel によってロードされている動的共有オブジェクトへのリファレンス.
`name`	モジュールの名前.
`nb`	モジュールの内部番号.
`version`	モジュールのバージョン番号.
`nbref`	モジュールを使っているモデルの数.

例 以下では，モジュールがモデルにロードされ，名前とバージョン番号を表示する.

```
XPRMdsolib lib;
const char *name;
int libv;
    ...
lib = XPRMgetnextdso(NULL);
XPRMgetdsoinfo(lib,&name,NULL,&libv,NULL);
printf("First module: %s version %d.%d.%d\n", name,
        libv/1000000, (libv/1000)-1000*(libv/1000000),
        libv - 1000*(libv/1000));
```

補足 `XPRMgetdsoinfo` はモジュールに関する情報を返します．バージョン番号は整数でコード化されています．例えば，1.2.3 は 1002003 になります．引数 `nbref` に 0 が返ってくれば，そのモジュールは現在使用されていません.

関連するトピック `EXAMINE`, `LSLIBS`

XPRMgetdsoparam

目的 現在の制御/属性の値（パラメータ）を返す.

概略 `int XPRMgetdsoparam(XPRMmodel model, XPRMdsolib dso,const char *name, int *type, XPRMalltypes *value);`

引数

model	現在のモデル
dso	Mosel によってロードされている動的共有オブジェクトへのリファレンス.
name	制御/属性の名前.
type	制御/属性の型.
value	制御/属性の返り値.

例 以下では，制御 VERBOSE の値に基づいてモジュール mmxprs が決定される.

```
XPRMmodel mod;
XPRMdsolib lib;
XPRMalltypes atbl;
XPRMboolean value;
int type;
    ...
if((lib = XPRMfinddso("mmxprs")) != NULL)
{
  XPRMgetdsoparam(mod,lib,"xprs_verbose",&type,&atbl);
  if(XPRM_TYP(type) == XPRM_TYP_BOOL)
  {
    value = atbl.boolean;
    printf("VERBOSE currently has the value '%s'\n",
          (value==1 ? "true" : "false"));
   }
  }
```

補足 XPRMgetdsoparam は，モデルが既に実行されてリクエストされたモジュールを使っている場合でないと実行できません.

関連するトピック EXAMINE，getparam，XPRMgetnextdsoparam

XPRMgetdual

目的 線形制約式の双対値を返す.

概略 `double XPRMgetdual(XPRMmodel model, XPRMlinctr ctr);`

引数

`model`	現在のモデル.
`ctr`	線形制約式へのリファレンス.

例 以下では，制約式 `first` の双対値が表示されている.

```
XPRMmodel model;
XPRMalltypes atctr;
XPRMlinctr first;
int nReturn;
    ...
XPRMrunmod(model,&nReturn,NULL);
XPRMfindident(model,"first",atctr);
first = ctr.linctr;
printf("%g\n",XPRMgetdual(model,first));
```

補足 `XPRMgetdual` は，問題がすでに解かれていて，制約式が問題に含まれている場合に，線形制約式の双対値を返します．そうでない場合は 0 です.

関連するトピック `getdual`, `XPRMgetact`, `XPRMgetcsol`, `XPRMgetslack`, `XPRSgetsol`（オプティマイザリファレンスマニュアル参照）

XPRMgetelsetndx

目的 集合の要素の添字を得る.

概略 `int XPRMgetelsetndx(XPRMmodel model, XPRMset set,  XPRMalltypes *elt);`

引数

`model`	モデルへのリファレンス.
`set`	集合へのリファレンス.
`elt`	要素へのリファレンス.

例 以下では，集合 `Items` の要素 2 の添字を表示している.

```
XPRMmodel model;
XPRMalltypes atsets, atelts;
XPRMset = items;
    ...
XPRMfindident(model,"Items",&atsets);
items = atsets.set;

atelts.integer = 2;
if(XPRMgetelsetndx(model,items,&atelts) >= 0)
  printf("Index of 2 in Items is %d\n",
         XPRMgetelsetndx(model,items,&atelts));
```

補足 負の値が返された場合は，集合にその要素は含まれていないことを示します.

関連するトピック `XPRMfirstsetndx`, `XPRMgetelsetval`, `XPRMgetsetsize`, `XPRMgetsettype`, `XPRMlastsetndx`

XPRMgetelsetval

目的 集合の要素の値を返す.

概略 `XPRMalltypes *XPRMgetelsetval(XPRMset set, int ind,XPRMalltypes *value);`

引数

`set`	集合へのリファレンス.
`ind`	添字番号.
`value`	返り値を返す場所へのポインタ.

例 以下の例では，集合 `Items` の要素が検索されてプリントされている.

```
XPRMmodel model;
XPRMalltypes atset, atelt;
XPRMset items;
...
XPRMfindident(model,"Items",&atset);
items = atset.set;
if(XPRM_TYP(XPRMgetsettype(items)) == XPRM_TYP_INT)
  for(i=1;i<=XPRMgetsetsize(items);i++)
  {
    XPRMgetelsetval(items,i,&atelt);
    printf("Items[%d] = %d\n",j,atelt.integer);
  }
```

補足 C の標準と異なり，添字は 0 ではなく 1 から始まります. 上の例を見てください.

関連するトピック `XPRMcmpindices`, `XPRMgetelsetndx`, `XPRMgetsetsize`, `XPRMgetsettype`

XPRMgetmodinfo

目的 モデルの情報を返します.

概略 `void XPRMgetmodinfo(XPRMmodel model, const char **name,int *number, const char **syscom, const char **usrcom,int *size);`

引数

model	現在のモデル.
name	モデルの名前を返す場所へのポインタ.
number	モデルの命令番号を返す場所へのポインタ.
syscom	システムのコメントを返す場所へのポインタ.
usercom	ユーザのコメントを返す場所へのポインタ.
size	モデルが使用しているメモリのサイズ（バイト）を返す場所へのポインタ.

例 以下では. XPRMgetmodinfo は問題の名前を検索するために使われている.

```
XPRMmodel model;
const char *name;
    ...
XPRMgetmodinfo(model,&name,NULL,NULL,NULL);
printf("Problem name is ‘ %s ’\n",name);
```

補足 対応する情報が必要ない場合は, model 以外の引数が NULL です.

関連するトピック INFO, LIST, XPRMcompmod (COMPILE), XPRMloadmod (LOAD)

XPRMgetnextdso

目的 次の動的共有オブジェクトを得る.

概略 `XPRMdsolib XPRMgetnextdso(XPRMdsolib dso);`

引数

`dso`	Mosel がロードしている動的共有オブジェクトへのリファレンスか `NULL`.

例 以下では，モデルがロードしているモジュールが返されて，名前とバージョンが表示されている.

```
XPRMdsolib lib;
const char *name;
int libv;
    ...
for(lib = XPRMgetnextdso(NULL); lib != NULL; lib =
          XPRMgetnextdso(lib))
{
  XPRMgetdsoinfo(lib,&name,NULL,&libv,NULL);
  printf("module: %s version %d.%d.%d\n", name,
        libv/1000000, (libv/1000)-1000*(libv/1000000),
        libv - 1000*(libv/1000));
}
```

補足 `XPRMgetnextdso` は Mosel がロードしているモジュールのリストの次のモジュールを返します．与えられたモジュールがリストの最後のときは，`NULL` を返し，リストの最初に戻ります.

関連するトピック `XPRMfinddso`, `XPRMgetdsoinfo`, `XPRMgetnextdsoconst`, `XPRMgetnextdsoparam`, `XPRMgetnextdsoproc`

XPRMgetnextdsoconst

目的 モジュールによって定義されている定数のリストの次の定数を返す.

概略 `void *XPRMgetnextdsoconst(XPRMdsolib dso, void *ref, const char **name, int *type, XPRMalltypes *value);`

引数

`dso`	Mosel によってロードされている動的共有オブジェクトへのリファレンス.
`ref`	リファレンスポインタか `NULL`.
`name`	返された定数の名前.
`type`	返された定数の型.
`value`	返された定数の値.

例 以下では，全てのロードされているモジュールの定数が表示される.

```
XPRMdsolib lib;
XPRMalltypes atval;
void *ref;
const char *name;
int type;
    ...
ref = NULL;
for(lib = XPRMgetnextdso(NULL); lib != NULL; lib =
XPRMgetnextdso(lib))
{
  while((ref = XPRMgetnextdsoconst(lib, ref, &name,
       &type, &atval)) != NULL)
    printf("%s\n",name);
}
```

補足 `XPRMgetnextdsoconst` は，与えられたモジュールによって定義された次の定数を返します．第 2 引数は，定数のテーブルの中での現在の場所を格納するのに使います．この引数が `NULL` の場合，テーブルの最初の定数が返されます．モジュールに定義された最後の定数のリファレンスの場合には，関数は `NULL` を返します．そうでなければ，返り値は入力引数を `ref` として使えます．定数の型と値に関する返された情報はモデルの識別子と同様にデコードされます（`XPRMfindident` 参照）.

関連するトピック `XPRMfindident`, `XPRMgetnextdso`, `XPRMgetnextdsoparam`, `XPRMgetnextdsoproc`, `XPRMgetnextdsotype`

XPRMgetnextdsoparam

目的 与えられたモジュールのリストの次の制御/属性 (パラメータ) を得る.

概略 `void *XPRMgetnextdsoparam(XPRMdsolib dso, void *ref,const char **name, const char **desc, int *type);`

引数

`dso`	Mosel によってロードされている動的共有オブジェクトへのリファレンス.
`ref`	リファレンスポインタか `NULL`.
`name`	返された制御/属性の名前.
`desc`	返された制御/属性の記述.
`value`	返された制御/属性の値.

例 以下では，全てのロードされているモジュールの制御と属性が検索されて出力ファイル `dsoparam.dat` に書き込まれている.

```
XPRMdsolib lib;
const char *libname, *name, *desc;
int type;
void *ref;
FILE *out;
...
out = fopen("dsoparams.dat","w");
lib = XPRMgetnextdso(NULL);
while(lib != NULL)
  {
   XPRMgetdsoinfo(lib,&libname,NULL,NULL,NULL);
   fprintf(out,"Parameters in %s:\n",libname);
   ref = XPRMgetnextdsoparam(lib,NULL,&name,&desc,&type);
   while(ref != NULL)
   {
     if(type & XPRM_CPAR_WRITE)
       fprintf(out,"\tcontrol:\t");
     else fprintf(out,"\tattribute:\t");
     if(XPRM_TYP(type) == XPRM_TYP_INT)
       fprintf(out,"integer\t\t");
     else if(XPRM_TYP(type) == XPRM_TYP_REAL)
       fprintf(out,"real\t\t");
     else if(XPRM_TYP(type) == XPRM_TYP_STRING)
       fprintf(out,"string\t\t");
```

```
      else if(XPRM_TYP(type) == XPRM_TYP_BOOL)
        fprintf(out,"boolean\t\t");
      fprintf(out,"%s\n",name);
      ref=XPRMgetnextdsoparam(lib,ref,&name,&desc,&type);
    }
    lib = XPRMgetnextdso(lib);
    fprintf(out,"\n");
  }
  fclose(out);
```

補足

1. XPRMgetnextdsoparam. の呼び出しでリファレンスポインタが返されます.
2. 第 2 引数は制御/属性のテーブルでの現在の位置を格納するのに使います. この引数が NULL の場合, テーブルの最初の制御/属性が返されます. NULL を返すのは, モジュールの最後の制御/属性のリファレンスで呼び出された場合です. そうでなければ, ref は次の制御/属性の入力引数として使うことができます.
3. 型はマクロ XPRM_TYP を使ってデコードできます. 更に, XPRM_CPAR_READXPRM_CPAR_WRITE はパラメータが (getparam と setparam を使って) 読んだり書いたりできるかどうかを表します. 引数 desc は制御/属性の関数の記述で, 必ずしも情報が得られるとは限りません (NULL かもしれません). 制御/関数の機能に必要とされるのは全てモジュールではないことに注意してください.

関連するトピック XPRMgetnextdso, XPRMgetnextdsoconst, XPRMgetnextdsoproc, XPRMgetnextdsotype

XPRMgetnextdsoproc

目的 与えられたモジュールのリストの次のサブルーチンを得る.

概略 `void *XPRMgetnextdsoproc(XPRMdsolib dso, void *ref,const char **name, const char **partyp, int *nbpar,int *type);`

引数

dso	Mosel によってロードされている動的共有オブジェクトへのリファレンス.
ref	リファレンスポインタか NULL.
name	返されたルーチン (手続きか関数) の名前.
partyp	返されたルーチンの引数の記述.
desc	返されたルーチンの引数の数.
type	返されたルーチンの結果の型.

例 以下では，モジュールで使える手続き/関数のリストとその引数を表示している.

```
XPRMdsolib lib;
const char *name, *partyp;
int nbpar, type;
void *ref;
    ...
ref = NULL;
while((ref = XPRMgetnextdsoproc(lib, ref, &name, &partyp,
      &nbpar, &type)) != NULL)
{
  printf("%s; argument types  '%s '\n",name,partyp);
}
```

補足

1. XPRMgetnextdsoproc は次の XPRMgetnextdsoproc の呼び出しへのリファレンスポインタを返します.
2. この関数は，与えられたモジュールで定義された次のサブルーチンを返します．第 2 の引数はサブルーチンのテーブル内の現在の場所を格納するのに使います．これが NULL ならば，テーブルの最初のルーチンに戻ります．与えられたモジュールで定義された最後のサブルーチンである場合，NULL を返します．そうでなければ，引数 ref は次のサブルーチンの入力引数として使うことができます．型と引数は，ネイティブな関数でない場合にはモデルの手続きと関数と同様にデコードできます (XPRMgetprocinfo 参照)．ネイティブな関数は，関数の型は XPRM_TYP_EXTN で，引数 partyp は関数の型に「：」の後に続いて始まります.（例えば，”mytype:|mytype| ” の場合，オブジェクトの型「mytype」を除いた関数の型「mytype」の署名記号です．) 引数の型が違う同じ名前のサブルーチンがある場合（オーバーロード），何度も同じ名前が返されることに注意してください.

関連するトピック XPRMgetnextdso, XPRMgetnextdsoconst, XPRMgetnextdsoparam, XPRMgetnextdsotype, XPRMgetprocinfo

XPRMgetnextdsotype

目的 与えられたモジュールのリストの次の型を得る.

概略 `void *XPRMgetnextdsotype(XPRMdsolib dso, void *ref,const char **name);`

引数

`dso`	Mosel によってロードされている動的共有オブジェクトへのリファレンス.
`ref`	リファレンスポインタか NULL.
`name`	返された型の名前.

例 以下では，ロードされているライブラリの全ての型があげられている.

```
XPRMdsolib lib;
const char *name, *typename;
void *type;
    ...
for(lib=XPRMgetnextdso(NULL); lib!=NULL;
        lib=XPRMgetnextdso(lib))
{
  XPRMgetdsoinfo(lib,&name,NULL,NULL,NULL);
  printf("Module %s is loaded\n",name);
  for(type=XPRMgetnextdsotype(lib,NULL,&typename);
        type!=NULL;
        type=XPRMgetnextdsotype(lib,type,&typename))
    printf("\t%s\n",typename);
}
```

補足 この関数は，与えられたモジュールで定義されている次の型の名前を返します．第 2 引数は型のテーブル内の現在の位置を格納するのに使われます．この引数の NULL である場合，テーブルの最初の方に戻ります．与えられたモジュールで定義された最後の型である場合，NULL を返します．そうでなければ，引数 `ref` は次の型の入力引数として使うことができます.

関連するトピック `XPRMgetnextdso`, `XPRMgetnextdsoconst`, `XPRMgetnextdsoparam`, `XPRMgetnextdsoproc`, `XPRMgettypename`

XPRMgetnextident

目的 与えられたモジュールのリストの次の識別子を得る.

概略 `const char *XPRMgetnextident(XPRMmodel model, void **ref);`

引数

`model`	現在のモデル.
`ref`	現在の位置が格納されている場所へのポインタ. この関数が呼ばれるたびに更新される. この引数が NULL の場合, テーブルの最初の識別子に戻る.

例 下の例は辞書の全ての識別子のリストを示すものである.

```
XPRMmodel model;
void *ref;
const char *name;
    ...
ref = NULL;
for(name = XPRMgetnextident(model,&ref); ref!=NULL;
          name=XPRMgetnextident(model,&ref))
  printf("%s\n",name);
```

補足 XPRMgetnextident は辞書から識別子を返します. 全ての識別子が返された場合, あるいは内部のテーブルにおける最後の識別子へのリファレンスで呼ばれた場合には NULL を返します.

関連するトピック XPRMfindident

XPRMgetnextmod

目的 次のモデルを返す.

概略 `XPRMmodel XPRMgetnextmod(XPRMmodel model);`

引数

`model`	現在のモデル.

例 以下では，ロードされているモデルの系列が順番に反復される.

```
XPRMmodel model;
int nReturn;
    ...
model = XPRMgetnextmod(NULL);
while(model != NULL)
{
  XPRMrunmod(model,&nReturn,NULL);
  model = XPRMgetnextmod(model);
}
```

補足 Mosel はロードされたモデルのリストを管理します．この関数は，内部のテーブルで与えられたモデルの次にあるモデルを返します．入力引数が `NULL` の場合，最初のモデルが返されます．最後のモデルで呼ばれた場合には，`NULL` を返します.

関連するトピック `XPRMfindmod`, `XPRMgetmodinfo`, `LIST`, `SELECT`

XPRMgetnextparam

目的 モデルの次のパラメータを返す.

概略 `const char *XPRMgetnextparam(XPRMmodel model, void **ref);`

引数

`model`	現在のモデル.
`ref`	現在の位置を格納する場所へのポインタ. この関数が呼ばれるたびに更新される. この引数が NULL の場合, モデルの最初のパラメータが返される.

例 以下では, モデルの全てのパラメータ（parameters ブロックで宣言された定数）のリストが表示されている.

```
XPRMmodel model;
void *ref;
const char *name;
    ...
ref = NULL;
for(name=XPRMgetnextparam(model,&ref); ref!=NULL;
        name=XPRMgetnextparam(model,&ref))
  printf("%s\n",name);
```

補足 この関数は, モデルの最後のパラメータへのリファレンスで呼ばれた場合には, 第 2 引数に NULL を返します.

関連するトピック XPRMgetnextdso, XPRMgetnextdsoparam

XPRMgetnextproc

目的 手続きか関数の次にオーバーロードされているバージョンを得る.

概略 `XPRMproc XPRMgetnextproc(XPRMproc proc);`

引数

proc	手続きか関数へのリファレンス.

例 以下では，関数/手続き `isPrime` の引数や返り値の型が調べられている.

```
XPRMmodel model;
XPRMalltypes atproc;
XPRMproc proc;
const char *partyp;
int nbpar, type;
    ...
XPRMfindident(model,"isPrime",&atproc);
proc = atarr.proc;
while(proc != NULL)
{
  XPRMgetprocinfo(proc,&partyp,&nbpar,&type);
  printf("isPrime can take %d arguments and ",nbpar);
  switch(type)
  {
    case XPRM_TYP_NOT: printf("is a procedure\n"); break;
    case XPRM_TYP_INT: printf("returns an int\n"); break;
    case XPRM_TYP_REAL: printf("returns a real\n");break;
    case XPRM_TYP_STRING: printf("returns a string\n");
    break;
    default: printf("returns a strictly Mosel type\n");
  }
  proc = XPRMgetnextproc(proc);
}
```

補足

1. `XPRMgetnextproc` は手続きか関数へのリファレンスを返します．オーバーロードされているサブルーチンが定義されていない場合には `NULL` を返します.
2. この関数は，与えられたサブルーチンの次にオーバーロードして定義されているサブルーチンを返します．モデルの中で，引数が異なるサブルーチンが何度も定義されていることがあります．この関数は，定義されているサブルーチンの全てのオーバーロードされたバージョンにアクセスします.

関連するトピック `XPRMgetnextdso`, `XPRMgetnextdsoproc`, `XPRMgetprocinfo`

XPRMgetobjval

目的 目的関数の値を返す.

概略 `double XPRMgetobjval(XPRMmodel model);`

引数

`model`	現在のモデル.

例 以下では，myprob がロードされて実行され，最適解が見つかった場合には目的関数の値を表示する.

```
XPRMmodel model;
int nReturn;
    ...
model = XPRMloadmod("","myprob",NULL,NULL);
XPRMrunmod(model,&nReturn,NULL);
if((XPRMgetprobstat(model) & XPRM_PBRES) == XPRM_PBOPT)
  printf("The maximum profit is %g\n",
         XPRMgetobjval(model));
```

関連するトピック DISPLAY, getobjval, XPRMgetcsol, XPRMgetdual, XPRMgetrcost, XPRMgetslack, XPRMgetvsol, LPOBJVAL(オプティマイザリファレンスマニュアル参照),MIPOBJVAL(オプティマイザリファレンスマニュアル参照)

XPRMgetprobstat

目的 モデルの問題の状態を返す.

概略 `int XPRMgetprobstat(XPRMmodel model);`

引数

`model`	現在のモデル.

例 以下では，解法ルーチンが終わった後に，問題の状態を標準出力に表示している.

```
switch(XPRMgetprobstat(model) & XPRM_PBRES)
{
  case XPRM_PBOPT: printf("Optimal solution found.\n");
                   break;
  case XPRM_PBINF: printf("Problem infeasible.\n");
                   break;
  case XPRM_PBUNB: printf("Problem is unbounded.\n");
                   break;
  default: printf("An error occurred in solution.\n");
                   break;
}
```

補足 XPRMgetprobstat は，与えられたモデルのメインの問題の状態を返します．問題が得られない場合には 0 を返します．問題の状態は以下のようにビットエンコードされています.

XPRM_PBCHG	オプティマイザにロードされた問題は適切でない.
XPRM_PBSOL	解が得られる.

解法の状態は問題の状態の XPRM_PBRES ビットをチェックすることによって得られます．可能な値は以下の通りです.

XPRM_PBCPT	最適解が見つかった
XPRM_PBUNF	最適化が終わっていない.
XPRM_PBINF	問題が実行できない.
XPRM_PBUNB	問題が制約されていない.
XPRM_PBOTH	最適化に失敗した.

関連するトピック getprobstat，LPSTATUS(オプティマイザリファレンスマニュアル参照) ,MIPSTATUS(オプティマイザリファレンスマニュアル参照)

XPRMgetprocinfo

目的 手続きか関数に関する情報を提供する.

概略 `int XPRMgetprocinfo(XPRMproc proc, const char **argtyp,int *nbargs, int *type);`

引数

proc	手続きか問題へのリファレンス.
srgtyp	返された引数の型をあらわす文字列.
nbargs	返された引数の数.
type	返された関数の型. モデルの他の識別子のようにデコードできる. 手続きは返り値がないことに注意 (type ＝ XPRM_TYP_NOT).

例 以下では, 関数/手続き isPrime の引数と返り値の型を調べている.

```
XPRMmodel model;
XPRMalltypes atproc;
XPRMproc proc;
const char *partyp;
int nbpar, type;
    ...
XPRMfindident(model,"isPrime",&atproc);
proc = atarr.proc;
while(proc != NULL)
{
  XPRMgetprocinfo(proc,&partyp,&nbpar,&type);
  printf("isPrime can take %d arguments and ",nbpar);
  switch(type)
  {
    case XPRM_TYP_NOT: printf("is a procedure\n"); break;
    case XPRM_TYP_INT: printf("returns an int\n"); break;
    case XPRM_TYP_REAL: printf("returns a real\n");break;
    case XPRM_TYP_STRING: printf("returns a string\n");
    break;
    default: printf("returns a strictly Mosel type\n");
  }
  proc = XPRMgetnextproc(proc);
}
```

補足 引数の型をあらわす文字列は, その関数の署名記号です. 文字列は以下のような文字からなります.

i	整数
r	実数
s	文字列
b	ブーリアン
v	決定変数 (mpvar 型)
c	線形制約式 (linctr 型)
I	範囲集合
a	配列（どんな種類でも）
e	集合（どんな種類でも）
—xxx—	「xxx」という名前の外部型
！xxx！	「xxx」という名前の集合
Andc.t	「ndx」で添字を与えられる型「t」の配列. 「ndx」は添字集合の型を表す.
Et	型「t」の集合

例えば，下のような手続きの署名記号は「iAIr!myset!.sb ’」です.

```
proc(n:integer, tab:array(range, set of real, myset) of string, flag:boolean)
```

関連するトピック XPRMgetmodinfo, XPRMgetnextdsoproc, XPRMgetnextproc

XPRMgetrcost

目的 変数のリデューストコストの値を返す.

概略 `double XPRMgetrcost(XPRMmodel model, XPRMmpvar var);`

引数

`model`	現在のモデル.
`var`	決定変数へのリファレンス.

例 以下では，解が見つかっていたら，変数 a のリデューストコストを返している.

```
XPRMmodel model;
XPRMalltypes atvar;
XPRMmpvar a;
    ...
if((XPRMgetprobstat(model) & XPRM_PBRES) == XPRM_PBOPT)
{
  XPRMfindident(model,"a",&atvar);
  a = atvar.mpvar;
  printf("a = %g\n", XPRMgetrcost(model,a));
}
```

補足 XPRMgetrcost は，問題がすでに解かれている場合には与えられた変数についてのリデューストコストを返します（そうでなければ 0 を返します.）.

関連するトピック getrcost，XPRMgetcsol，XPRMgetdual，XPRMgetobjval，XPRMgetvsol，XPRSgetsol（オプティマイザリファレンスマニュアル参照）

XPRMgetsetsize

目的 集合のサイズ（集合の要素数）を返す.

概略 `int XPRMgetsetsize(XPRMset set);`

引数

`set`	集合へのリファレンス.

例 以下の例では，集合 `Items` の要素が検索されて表示されている.

```
XPRMmodel model;
XPRMalltypes atset, atelt;
XPRMset items;
...
XPRMfindident(model,"Items",&atset);
items = atset.set;
if(XPRM_TYP(XPRMgetsettype(items)) == XPRM_TYP_INT)
  for(i=1;i<=XPRMgetsetsize(items);i++)
  {
    XPRMgetelsetval(items,i,&atelt);
    printf("Items[%d] = %d\n",j,atelt.integer);
  }
```

関連するトピック `XPRMgetelsetndx, XPRMgetelsetval, XPRMgetsettype`

XPRMgetsettype

目的 集合の型を得る.

概略 `int XPRMgetsettype(XPRMset *set);`

引数

`set`	集合へのリファレンス.

例 以下の例では，集合 `Items` の要素が検索されて表示されている.

```
XPRMmodel model;
XPRMalltypes atset, atelt;
XPRMset items;
   ...
XPRMfindident(model,"Items",&atset);
items = atset.set;
if(XPRM_TYP(XPRMgetsettype(items)) == XPRM_TYP_INT)
  for(i=1;i<=XPRMgetsetsize(items);i++)
  {
    XPRMgetelsetval(items,i,&atelt);
    printf("Items[%d] = %d\n",j,atelt.integer);
  }
```

補足 集合の型は，集合の全ての要素の型と，集合の記憶クラスです．要素の型はマクロ `XPRM_TYP(type)` を使って抽出できます．記憶クラスはマクロ `XPRM_GRP(type)` を使って抽出できます．`XPRM_GRP_GEN` ビットがセットされていれば，範囲集合ではありません．`XPRM_GRP_DYN` ビットがセットされていれば，集合は動的で，拡張されることがあります.

関連するトピック `XPRMgetelsetndx`, `XPRMgetelsetval`, `XPRMgetsetsize`

XPRMgetslack

目的 線形制約式のスラック値を返す.

概略 `double XPRMgetslack(XPRMmodel model, XPRMlinctr ctr);`

引数

`model`	現在のモデル.
`ctr`	線形制約式へのリファレンス.

例 以下は，ロードされたモデルを実行し，制約式 `first` のスラック値を表示している.

```
XPRMmodel model;
XPRMalltypes atctr;
XPRMlinctr first;
int nReturn;
    ...
XPRMrunmod(model,&nReturn,NULL);
XPRMfindident(model,"first",&atctr);
first = atctr.linctr;
printf("%g\n",XPRMgetslack(model,first));
```

補足 `XPRMgetslack` は，問題がすでに解かれている場合，与えられた線形制約式のスラック値を返します. そうでない場合は 0 を返します.

関連するトピック `getslack`, `XPRMgetcsol`, `XPRMgetdual`, `XPRMgetobjval`, `XPRMgetrcost`, `XPRMgetvsol`, `XPRSgetsol`(オプティマイザリファレンスマニュアル参照)

XPRMgettypename

目的 外部方のコードに対応する名前を返す.

概略 `const char *XPRMgettypename(XPRMmodel model, int type);`

引数

`model`	モデルへのリファレンス.
`type`	外部型のコード.

関連するトピック `XPRMgetnextdsotype`

XPRMgetvarnum

目的 決定変数の列の数を得る.

概略 `int XPRMgetvarnum(XPRMmpvar var);`

引数

`var`	変数への参照.

例 以下は，変数 a と b の列の数を表示する例である.

```
XPRMmodel mod;
XPRMalltypes atvar;
XPRMmpvar a,b;
    ...
XPRMfindident(mod,"a",&atvar);
a = atvar.mpvar;
XPRMfindident(mod,"b",&atvar);
b = atvar.mpvar;
printf("variable 'a' has column number %d\n
        variable 'b' has column number %d\n",
        XPRMgetvarnum(a), XPRMgetvarnum(b));
```

補足 `XPRMgetvarnum` は，決定変数の列の数 (≥ 0) を返します．問題が得られなかったか，その変数がメインの問題になかった場合，負の値を返します.

関連するトピック `getvars`，`XPRMgetctrnum`，`XPRSgetcols`（オプティマイザリファレンスマニュアル参照）

XPRMgetversion

目的 Mosel のバージョンを文字列で返す.

概略 `const char *XPRMgetversion(void);`

引数 なし.

例 以下では，Mosel のバージョンが表示される.

```
XPRMinit();
version = XPRMgetversion();
printf("Mosel version %s\n",version);
```

関連するトピック `VERSION`（オプティマイザリファレンスマニュアル参照）,`XPRBgetversion`(BCL リファレンスマニュアル参照)

XPRMgetvsol

目的 変数の解の値を返す.

概略 `double XPRMgetvsol(XPRMmodel model, XPRMmpvar var);`

引数

`model`	現在のモデル.
`var`	決定変数へのリファレンス.

例 この例では，モデルの決定変数 a の値が表示される.

```
XPRMmodel model;
XPRMalltypes atvar;
XPRMmpvar a;
    ...
XPRMfindident(model,"a",&atvar)
a = atvar.mpvar
printf("a = %g\n",XPRMgetvsol(model,a));
```

補足 XPRMgetvsol は，問題がすでに解かれている場合に与えられた変数の値を返します (LP では，LP 解か 0，グローバルでは最後の整数解か 0).

関連するトピック DISPLAY, getsol, XPRMgetcsol, XPRMgetdual, XPRMgetobjval,XPRMgetrcost, XPRMgetslack, XPRSgetsol（オプティマイザリファレンスマニュアル参照）

INFO

目的 アクティブな問題に関する情報を表示する.

概略 INFO *[symbol]*

引数

symbol	アクティブな問題の記号.

例 以下の例は，Mosel モデル myprob.mos がコンパイル，ロードされて実行されている．記号 profit と myproc の情報が尋ねられている.

```
CLOAD myprob
RUN
INFO profit
INFO myproc
```

補足 引数なしでは，このコマンドは現在実行している問題に関する情報を表示します（これは問題の報告に有益です）．引数は現在のモデルの中の記号として解釈されます．要求された記号が存在する場合，このコマンドは型と構造に関する情報を表示します.

関連するトピック DISPLAY，XPRMgetmodinfo，XPRMgetprocinfo，SYMBOLS

XPRMinit

目的 Mosel を初期化する．この関数は他のライブラリ関数が実行される前に呼び出す必要がある．

概略 `int XPRMinit(void);`

引数 なし．

例 この例では，モデルのロードと実行は初期化が成功したときだけ実行される．

```
XPRMmodel mod;
int nReturn;
if(XPRMinit())
{
  printf("Could not initialize Mosel.\n");
}
else
{
  mod = XPRMloadmod("myprob.bim",NULL);
  XPRMrunmod(mod,&nReturn,NULL);
  XPRMfree();
}
```

補足 ライブラリ関数の呼び出しでは，エラーのチェックが常にされますが，エラーの大部分は初期化の段階で起こるので，初期化についてはこれは特に重要です．返り値が 32 である場合，ライセンスが見つからないのでアプリケーションは「トライアルモード」で実行されます．返り値が 1 である場合には，初期化に失敗しています．

関連するトピック `XPRMfree`，`XPRSinit`（オプティマイザリファレンスマニュアル参照）

XPRMisrunmod

目的 モデルが実行されているかどうかをチェックする.

概略 `int XPRMisrunmod(XPRMmodel model);`

引数

`model`	現在のモデル.

例 ソースファイルが以下のような内容を含んでいる場合,

```
#include <signal.h>
#include "xprm_rt.h"
void end_run(int sig)
{
  if(XPRMisrunmod(mod)) XPRMstoprunmod(mod);
}
```

メイン関数は, `CPRMrunmod` コマンドを実行する前に以下のような行を含む場合, CTRL-C キーでモデルの実行を中止することができる.

```
signal(SIGINT,end_run);
```

関連するトピック `XPRMrunmod (RUN), XPRMstoprunmod`

XPRMlastarrentry

目的 配列の最後のエントリの添字を得る.

概略 `int XPRMlastarrentry(XPRMarray array, int indices[]);`

引数

`array`	配列へのリファレンス.
`indices`	配列 `array` の次元が n であるとき，添字の n-タプル.

例 この例では，配列 `A` の最後のエントリの添字が表示される.

```
XPRMmodel model;
XPRMalltypes atarr;
XPRMarray A;
int i,*indices;
    ...
XPRMfindident(model,"A",&atarr);
A = atarr.array;
indices = malloc(XPRMgetarrdim(A)*sizeof(int));
XPRMlastarrentry(A,indices);
for(i=0;i<XPRMgetarrdim(A);i++)
  printf("Index of last entry of A in dimension %d is
       %d\n",i+1,indices[i]);
```

関連するトピック `XPRMfirstarrentry`, `XPRMfirstarrtruentry`, `XPRMnextarrentry`, `XPRMnextarrtruentry`

XPRMlastsetndx

目的 集合の最後の要素の添字を返す.

概略 `int XPRMlastsetndx(XPRMset set);`

引数

`set`	集合へのリファレンス.

例 この例では，集合 `Items` の要素が検索されて表示される.

```
XPRMmodel model;
XPRMalltypes atset, atelt;
XPRMset items;
   ...
XPRMfindident(model,"Items",&atset);
items = atset.set;
if((XPRMgetsettype(items) & XPRM_TYP_INT) == 0)
  for(i=firstsetndx(items);i<=XPRMlastsetndx(items);i++)
  {
    XPRMgetelsetval(items,i,&atelt);
    printf("Items[%d] = %d\n",j,atelt.integer);
  }
```

補足

1. 範囲集合の場合，一番大きい値（範囲の中の最大値）が返されます．string の集合では，最後の要素の添字は集合の要素数に対応します.
2. この関数を呼ぶ前に，集合が空かどうかを調べる (`XPRMgetsetsize` を使う) ことが望ましいです.

関連するトピック `XPRMfirstsetndx`, `XPRMgetelsetndx`, `XPRMgetelsetval`, `XPRMgetsetsize`

LIST

目的 現在ロードされている全てのモジュールのリストを表示する.

概略 LIST

引数 なし.

例 以下では，2 つのモデルファイル simple.bim と alterf.bim がロードされ，全てのロードされているモジュールがリストされ，アクティブな問題として「simple」が選ばれて実行されている.

```
LOAD simple
LOAD altered
LIST
SELECT simple
RUN
```

補足 各モデルで表示される情報は以下の通りです.

name	モデルの名前（ソースファイルの中で model 文で与えられる）.
number	モデル番号. ロードされたときに自動的に割り当てられる.
size	モデルが使っているメモリの量（バイト）.
system comment	コンパイラによって生成されたソースファイルの名前を示す文字列と もし含まれていれば，デバッグ情報と/または記号.
user comment	コンパイル時にユーザが定義したコメント. （CLOAD，XPRMcompmod (COMPILE) 参照）

アクティブな問題は名前の前にアスタリスク「＊」がマークされます. デフォルトでは最後にロードされた問題がアクティブです.

関連するトピック DELETE，XPRMgetnextmod，XPRMloadmod (LOAD)，SELECT，XPRMunloadmod

XPRMloadmod LOAD

目的 バイナリモデルファイルをロードする.

概略 `XPRMmodel XPRMloadmod(const char *filename, const char *intname);`
`LOAD` *filename*

引数

`filename`	バイナリモデルファイルの名前.
`intname`	内部の名前. 必要ない場合は `NULL`.

例 1(ライブラリ) 以下は，通常のモデルのロードと実行の仕方を示している.

```
XPRMmodel model;
int nReturn;
    ...
XPRMinit();
model = XPRMloadmod("myprob.bim",NULL);
XPRMrunmod(model,&nReturn,NULL);
XPRMfree();
```

例 2(コンソール) この例では，コンソールでモデル `myprob.bim` がロードされて実行されている.

```
LOAD myprob
RUN
QUIT
```

補足

1. `XPRMloadmod` はロードされたモデルへのポインタを返します. 成功でない場合は `NULL` を返します.
2. モデルをファイルからロードするとき，Mosel は自動的にモデルが必要とする追加のネイティブなモジュールを開きます. 内部の名前が提供されていれば，`.bim` ファイルが格納されている場所として使われます.
3. 既にコアメモリに存在するモデル（同じ内部の名前で）を 2 度目にロードすると，最初のモデルは消されて新しく作られたモデルへのリファレンスが返されます.

関連するトピック `CLOAD`, `XPRMcompmod` (`COMPILE`), `XPRMrunmod` (`RUN`)

LSLIB

目的 ロードされている全ての動的共有オブジェクト（DSO）の値のリストを，各モジュールのバージョンと参照の数（いくつのモジュールがそれを使っているか）とともに表示する．

概略 LSLIBS

引数 なし．

例 以下では，Mosel モデル myprob.mos がコンパイルされ，Mosel にロードされて実行されている．それから，Mosel はロードされたモジュールを問われている．

```
CLOAD myprob
RUN
LSLIBS
```

関連するトピック EXAMINE, XPRMflushdso (FLUSHLIBS), XPRMgetnextdso

XPRMnextarrentry

目的 配列の次のエントリの添字のリストを得る.

概略 `int XPRMnextarrentry(XPRMarray array, int indices[]);`

引数

`array`	配列へのリファレンス.
`indices`	配列 `array` の次元が n であるとき，添字の n-タプル.

例 以下では，2 次元配列 `A` の全ての添字のペアが決定されて，論理順に表示されている.

```
XPRMmodel model
XPRMalltypes atarr;
XPRMarray A;
XPRMset Asets[2];
int indices[2],i,j;
    ...
XPRMfindident(model,"A",&atarr);
A = atarr.array;
XPRMgetarrsets(A,Asets);
XPRMfirstarrentry(A,indices);
for(i=0;i<XPRMgetsetsize(Asets[0]);i++)
  for(j=0;j<XPRMgetsetsize(Asets[1]);j++)
  {
    printf("(%d,%d)\n",indices[0],indices[1]);
    XPRMnextarrentry(A,indices);
  }
```

補足 この関数は，与えられた配列の与えられたタプルに続くエントリの添字タプルを返します．配列の次のエントリは，最後のタプルの添字を先に列挙することで決定できます．引数 `indices` は，入力であり返り値でもあります．これは，関数によって，入力されたタプルの次の配列のエントリに対応するタプルに変えられます.

関連するトピック `XPRMfirstarrentry`, `XPRMfirstarrtruentry`, `XPRMnextarrentry_trs`, `XPRMnextarrtruentry`, `XPRMlastarrentry`

XPRMnextarrentry_trs

目的 転置された配列の次のエントリの添字のリストを得る.

概略 `int XPRMnextarrentry_trs(XPRMarray array, int indices[]);`

引数

`array`	配列へのリファレンス.
`indices`	配列 `array` の次元が n であるとき，添字の n-タプル.

例 以下では，2 次元配列 `A` の全ての添字のペアが決定され，転置の配列に現れる順に表示されている.

```
XPRMmodel model
XPRMalltypes atarr;
XPRMarray A;
XPRMset Asets[2];
int indices[2],i,j;
...
XPRMfindident(model,"A",&atarr);
A = atarr.array;
XPRMgetarrsets(A,Asets);
XPRMfirstarrentry(A,indices);
for(i=0;i<XPRMgetsetsize(Asets[0]);i++)
  for(j=0;j<XPRMgetsetsize(Asets[1]);j++)
  {
    printf("(%d,%d)\n",indices[0],indices[1]);
    XPRMnextarrentry_trs(A,indices);
  }
```

補足

1. この関数は，与えられた配列の転置において，与えられたタプルに続くエントリの添字タプルを返します．転置された配列の次のエントリは，最初のタプルの添字を先に列挙することで決定できます.
2. 引数 `indices` は，入力であり，返り値でもあります．これは，関数によって，入力されたタプルの次の配列のエントリに対応するタプルに変えられます.

関連するトピック `XPRMfirstarrentry`, `XPRMfirstarrtruentry`, `XPRMnextarrentry`, `XPRMnextarrtruentry`, `XPRMlastarrentry`

XPRMnextarrtruentry

目的 配列の次の真のエントリの添字のリストを得る.

概略 `int XPRMnextarrtruentry(XPRMarray array, int indices[]);`

引数

`array`	配列へのリファレンス.
`indices`	配列 `array` の次元が n であるとき，添字の n-タプル.

例 以下では，2 次元配列 `A` の全ての真のエントリの添字が決定されて，論理順に表示されている.

```
XPRMmodel model;
XPRMalltypes atarr;
XPRMarray A;
int indices[2];
    ...
XPRMfindident(model,"A",&atarr);
A = atarr.array;
XPRMfirstarrtruentry(A,indices);
for(i=0;i<XPRMgetarrsize(A);i++)
{
  printf("(%d,%d)\n",indices[0],indices[1]);
  XPRMnextarrtruentry(A,indices);
}
```

補足 与えられた配列の大きさが固定ならば（密な配列），この関数は `XPRMnextarrentry` と同じ働きをします. 動的な配列の場合，次の真のエントリの添字タプルを返します.

関連するトピック `XPRMfirstarrentry`, `XPRMfirstarrtruentry`, `XPRMnextarrentry`, `XPRMnextarrentry_trs`, `XPRMlastarrentry`

XPRMpreloaddso

目的 指定されたモジュールを明示的にロードする.

概略 `XPRMdsolib XPRMpreloaddso(const char *libname);`

引数

`libname`	ロードするモジュールの名前.

例 以下では，モジュール `mmodbc` がロードされ，名前とバージョンが表示されている.

```
XPRMdsolib lib;
const char *libname;
int libv;
    ...
XPRMpreloaddso("mmodbc");
lib = XPRMgetnextdso(NULL);
while(lib != NULL)
{
  XPRMgetdsoinfo(lib,&libname,NULL,&libv,NULL);
  printf("Module %s, version %d.%d.%d loaded\n", libname,
        libv/1000000, (libv/1000)-1000*(libv/1000000),
        libv - 1000*(libv/1000));
}
```

補足

1. モジュールのロードに成功した場合には，DSO ディスクリプタへのリファレンスを返し，そうでない場合は `NULL` を返します.
2. Mosel は，必要な場合にはオン・デマンドでコアメモリにモジュールをロードします．しかし，この関数を使ってモジュールをロードすることも可能です．既にメモリにモジュールにロードされている場合は何も動作はせず，対応する DSO ポインタが返されます.

関連するトピック `fflush`, `uses`, `XPRMautounloaddso`, `XPRMflushdso` (`FLUSHLIBS`)

QUIT

目的 現在の Mosel のセッションを終了する.

概略 QUIT

引数 なし.

例 下の例は，バイナリモデル myprob.bim をロードして実行し，目的関数の値を表示し (profit)，セッションを終わらせます.

```
LOAD myprob
RUN
DISPLAY profit
QUIT
```

関連するトピック exit, XPRMfree, RESET

RESET

目的 アクティブなモデルが使っている全てのリソースを開放して，アクティブなモデルを再初期化する．

概略 RESET

引数 なし．

例 下の例は，バイナリモデル bigprob.bim をロードして実行し，newprob.bim をロードする前にリソースを開放している．

```
LOAD bigprob
RUN
RESET
LOAD newprob
```

関連するトピック XPRMfree, QUIT

XPRMrunmod RUN

目的 モデルを実行する.

概略 `int XPRMrunmod(XPRMmodel model, int *returned, const char *parlist);`
`RUN` *[parlist]*

引数

`model`	現在のモデル.
`returned`	結果の返り値の場所へのポインタ.
`parlist`	カンマで区切られたモデルパラメータの初期化の文字列. `NULL` のこともある.

例 1(ライブラリ) 以下は，通常のモデルのロードと実行のやり方を示している.

```
XPRMmodel model;
int nReturn;
   ...
XPRMinit();
model = XPRMloadmod("myprob.bim",NULL);
XPRMrunmod(model,&nReturn,NULL);
XPRMfree();
```

例 2(コンソール) 以下では，Mosel モデル `myprob.mos` をコンパイル，ロード，実行している.

```
COMPILE myprob
LOAD myprob
RUN
QUIT
```

補足
1. 返り値は実行の状態で，以下のような値をとります.
   -1 モデルは実行されていない.
   0 正常終了.
   ≥ 0 実行中にエラーが起こった.
2. `XPRM_RT_STOP` ビットは，モデルの実行が関数 `XPRMstoprunmod` . によって中止された場合にはセットされます.

関連するトピック `CLOAD`, `DELETE`, `XPRMcompmod (COMPILE)`, `XPRMisrunmod`, `XPRMloadmod (LOAD)`, `XPRMstoprunmod`, `XPRMunloadmod`

SELECT

目的 モデルをアクティブな問題として選ぶ.

概略 SELECT *[number — name]*

引数

number	モデルの番号．モデルがロードされたときに自動的に割り当てられる．
name	モデルの名前．ソースファイルの model 文で与えられたもの．

例 以下では，モデル simple.bim と alterd.bim がロードされ,「alterd」が実行され，次に「select」が実行される.

```
LOAD simple
LOAD altered
RUN
SELECT simple
RUN
```

補足 モデルは名前か命令番号を用いて選ぶことができます．モデルのリファレンスが提供されない場合，現在のアクティブなモデルの情報が表示されます.

関連するトピック CLOAD，DELETE，LIST，XPRMloadmod (LOAD)，XPRMunloadmod

XPRMstoprunmod

目的 実行中のモデルを停止する.

概略 `void XPRMstoprunmod(XPRMmodel model);`

引数

`model`	現在のモデル.

例 ソースファイルに以下が含まれている場合には,

```
#include <signal.h>
#include "xprm_rt.h"
void end_run(int sig)
{
  if(XPRMisrunmod(mod)) XPRMstoprunmod(mod);
}
```

メイン関数は, `CPRMrunmod` コマンドを実行する前に以下のような行を含む場合 CTRL-C キーでモデルの実行を中止することができる.

```
signal(SIGINT,end_run);
```

関連するトピック `DELETE`, `XPRMisrunmod`, `XPRMloadmod (LOAD)`, `XPRMrunmod (RUN)`, `XPRMunloadmod`

SYMBOL

目的 アクティブな問題によって講評される記号のリストを表示する.

概略 SYMBOL *[-options]*

引数

options		どのような記号を表示するかをフィルタするオプションは以下の通りである.
	c	定数を表示.
	s	関数/手続き（サブルーチン）を表示.
	p	制御/属性（パラメータ）を表示.
	o	全てを表示.

例 以下では，Mosel モデル myprob.mos をコンパイル，ロードし，公表されている記号を表示してから実行している.

```
CLOAD myprob
SYMBOLS
RUN
```

関連するトピック DISPLAY, EXAMINE, INFO

SYSTEM

目的 OS のコマンドを実行する.

概略 `SYSTEM` *command*

引数

`command` 実行するコマンド.

例 以下では，Windows マシンで Mosel モデル `simple.mos` をコンパイル，ロード，実行し,「simple」の作業ディレクトリを表示する.

```
CLOAD simple
RUN
SYSTEM 'dir simple.*'
```

関連するトピック `system`

XPRMunloadmod

目的 モデルが使っている全てのリソースを開放し，モデルをアンロードする．

概略 `int XPRMunloadmod(XPRMmodel model);`

引数

`model`	現在のモデル．

例 以下は，モデルがロードされ，実行されてからアンロードされる例を示す．

```
XPRMmodel model;
int nReturn;
    ...
if(XPRMinit()) printf("Could not initialize.\n");
else
{
  model = XPRMloadmod("myprob.bim",NULL);
  XPRMrunmod(model,&nReturn,NULL);
  XPRMunloadmod(model)
  XPRMfree();
}
```

補足 `XPRMunloadmod` は，モデルを実行する前の場合は失敗します．

関連するトピック `DELETE`, `XPRMisrunmod`, `XPRMloadmod (LOAD)`, `XPRMrunmod (RUN)`, `XPRMstoprunmod`

第 6 章

制御パラメータ

Mosel では，多数の制御パラメータを検索し，モデルの解法のプロセスに影響を与えるために，制御パラメータの値を新しい値に変更することができます．Mosel 自身が制御の集合を持っているだけでなく，たくさんのモジュールがより広い範囲の制御を可能にします．そのような外部の制御にアクセスする場合には，以下のような点に注意する必要があります．

- 言語のコアに含まれていないような制御にアクセスする場合には，関連するモジュールを「uses *module_name*」のようにしてロードしておく必要があります．
- 制御がどのモジュールに属するかについては，制御の名前にモジュールの名前が前置されているのでわかります．(例えば，mmodbc.SQLCOLSIZE)
- オプティマイザモジュール mmxprs の全ての制御は，XPRS_を前置する必要があります．(例えば，XPRS_LOADNAME, XPRS_VERBOSE．) 同様に，オプティマイザリファレンスマニュアルに説明されている制御から Mosel にアクセスする場合には，XPRS_PRESOLVE のようにします．

6.1 制御パラメータの検索と変更

Mosel 言語は，制御パラメータの値を検索してセットするために 2 つのコマンド getparam と setparam をサポートしています．これらを用いて，モデルが実行されるときはいつでもパラメータを変更し，モデルファイルに影響を与えるように埋め込むことができます．

Mosel ライブラリのユーザーのために，モデルの実行中，または，モデルの実行に引き続き，制御パラメータを検索するコマンドがあります (XPRMgetnextparam)．動的共有オブジェクト (DSO) として実行されたモジュールに関しては，コマンド XPRMgetdsoparam と XPRMgetnextdsoparam を用いて，各モジュールの全ての制御へのアクセスを行います．制御の設定のためのライブラリコマンドは存在しません．

IOCTRL

概要	インタープリタが IO エラーを無視するかどうかを表す.
型	boolean
値	`true` IO エラーを無視. `false` IO エラーを無視しない.
デフォルト値	`false`
モジュール	コア言語
関連ルーチン	`fclose,fopen,fskipline,read, readln, write, writeln`

LOADNAMES

概要	MPS 名をオプティマイザにロードするかどうかを表す.
型	boolean
値	`true` 名前をオプティマイザにロードするのを許可. `false` 名前をオプティマイザにロードするのを許可しない.
デフォルト値	`false`
モジュール	`mmxprs`
注意	オプティマイザ制御では, `getparam(XPRS_LOADNAMES)` のようにして `XPRS_`プレフィックスをつけて使う.
関連ルーチン	`loadprob, maximize, minimize`

REALFMT

概要	実数値のデフォルト C プリント形式.
型	string
デフォルト値	`"%g"`
モジュール	コア言語
関連ルーチン	`write, writeln`

SQLCOLSIZE

概要	データの交換のための許容される文字列の最大の長さ．これを越えたものはカットされる．
型	integer
最小値	8
最大値	1024
デフォルト値	64
モジュール	mmodbc
関連ルーチン	SQLexecute, SQLreadinteger, SQLreadreal, SQLreadstring

SQLCONNECTION

概要	アクティブな ODBC 接続を識別するための数．この値が変わることによって同時にいくつかの接続が可能になる．
型	integer
モジュール	SQLconnect によってセット
関連ルーチン	SQLdisconnect, SQLexecute, SQLreadinteger, SQLreadreal, SQLreadstring

SQLNDXCOL

概要	各行の最初の列が添字として解釈されるかどうかを表す．この値を false にすると，例えば，密なスプレットシートのテーブルのようなリレーショナルでないテーブルにアクセスするのに役立つ．
型	boolean
値	true 各行の第 1 列を添字として解釈． false 各行の第 1 列を添字として解釈しない．
デフォルト値	true
モジュール	mmodbc
関連ルーチン	SQLexecute, SQLreadinteger, SQLreadreal, SQLreadstring

SQLVERBOSE

概要 ODBC ドライバによるメッセージのプリントを有効にするかどうかを表す.
型 boolean
値 `true` メッセージの表示を有効にする.
`false` メッセージの表示を無効にする.
デフォルト値 `false`
モジュール `mmodbc`
関連ルーチン `SQLconnect`, `SQLdisconnect`, `SQLexecute`, `SQLreadinteger`, `SQLreadreal`, `SQLreadstring`

VERBOSE

概要 オプティマイザによるメッセージのプリントを有効にするかどうかを表す.
値 `true` メッセージの表示を有効にする.
`false` メッセージの表示を無効にする.
デフォルト値 `false`
モジュール `mmxprs`
注意 オプティマイザ制御では, `getparam(XPRS_VERBOSE)` のようにして `XPRS_`プレフィックスをつけて使う.
関連ルーチン `mmxprs` モジュールの全てのルーチン.

ZEROTOL

概要 実数同士を比較するときの零許容範囲.
型 real
デフォルト値 $1.0E-6$
モジュール コア言語.
関連ルーチン

第 7 章

問題の属性

Mosel では，モデル化および解法のプロセスの間に，ユーザーが特定の問題の多数の性質にアクセスすることができます．Mosel はそのような属性をそれ自身で持っているだけでなく，いくつかのモジュールによってサポートする様々な追加の情報を提供します．そのような外部の属性にアクセスしているときには，以下のようなことに注意しなければなりません．

- コア言語以外の属性にアクセスする場合には，関連するモジュールを「`uses` *module_name*」のようにしてロードしておく必要があります．
- 属性がどのモジュールに属するかについては，属性の名前にモジュールの名前が前置されているのでわかります．(例えば，`mmodbc.SQLROWCNT`)
- オプティマイザモジュール `mmxprs` の全ての属性は，`XPRS_`を前置する必要があります．(例えば，`XPRS_PROBLEM`．) 同様に，オプティマイザリファレンスマニュアルに説明されている属性から Mosel にアクセスする場合には，`XPRS_LPOJVAL` のようにします．

7.1 問題の属性の検索

Mosel 言語は，問題属性の値を検索するためにコマンド `getparam` をサポートしています．これを用いて，属性の値はモデルファイルの中で検索することができ，モデルファイルが実行されるとき，そのモデルがどのように組み立てられるかに影響を与えるために使うことができます．

Mosel ライブラリのユーザーのために，モデルの実行中，または，モデルの実行に引き続き，属性を検索するコマンドがあります (`XPRMgetnextparam`)．動的共有オブジェクト (DSO) として実行されたモジュールに関しては，コマンド `XPRMgetdsoparam` と `XPRMgetnextdsoparam` を使って各モジュールの全ての属性へのアクセスを行います．

IOSTATUS

概要	直前の IO 操作の状態.
型	integer
モジュール	コア言語
注意	このパラメータは，getparam で値を読まれた後に自動的にリセットされる.
セットするルーチン	fclose，fopen，fskipline，read，readln，write，writeln

NBREAD

概要	直線の手続き read または readln によって認識された要素の数.
型	integer
モジュール	コア言語
セットするルーチン	read,readln

PROBLEM

概要	オプティマイザ問題ポインタ．この属性は，C レベルで Mosel とオプティマイザの両方を使い場合にのみ必要となる.
型	integer
モジュール	mmxprs
注意	全てのオプティマイザ制御と同様，これは XPRS_という接頭辞をとる．使い方は，getparam(XPRS_PROBLEM) のようになる.
セットするルーチン	loadprob，maximize，minimize

SQLROWCNT

概要	直前の SQL コマンドによって影響された行の数.
型	integer
モジュール	mmodbc
セットするルーチン	SQLexecute，SQLreadinteger，SQLreadreal，SQLreadstring

SQLROWXFR

概要	直前の SQL コマンドの間に転送された行の数.
型	integer
モジュール	mmodbc
セットするルーチン	SQLexecute，SQLreadinteger，SQLreadreal，SQLreadstring

SQLSUCCESS

概要	直前の SQL コマンドの実行に成功したかどうかを示す.
型	Boolean
モジュール	`mmodbc`
セットするルーチン	`SQLconnect`, `SQLdisconnect`, `SQLexecute`, `SQLreadinteger`, `SQLreadreal`, `SQLreadstring`

付録 A

使用例

A.1 カットの生成

カット平面法は，整数解の凸包のカットオフされた部分に制約を与えるもので，LP 緩和の解法から，整数の実行可能な解答を引き出し，緩和された問題の解の限度を向上させます．Xpress-Optimizer は、自動的にカットの生成を行う (詳細はオプティマイザリファレンスマニュアル参照．) 生成されるカットの効果を示すために，自動的なカット生成を無効にします．

例題 大きい会社が，オフィスの掃除をアウトソースする計画を立てています．会社のオフィスサイトは $NSITES$ あり，$NAREAS$ エリアにグループ化されます．いくつかの専門的な掃除会社 (総数は $NCONTRACTORS$) はが異なるサイトに入札しました．コスト 0 は，サイトに入札していないことを意味します．

1 つの会社だけに依存するのを避けるため，隣接のエリアは異なる会社に割当てることにします．各サイト s(s は 1 から $NSITES$ の間) は，1 つの会社に割り当てますが，エリア a ごとに $LOWCON_a$ と $UPPCON_a$ の間の会社があります．

問題は以下のようにまとめることができます．

目的： 全ての定数についてのトータルのコストを最小化する．

制約： 各サイトは必ず 1 社が掃除をしなければならない．隣接するエリアは同じ会社が担当してはいけない．1 つのエリアごとに，会社の数の上限と下限が決まっている．

問題を数学的に定式化するため，2 つの種類の決定変数を使います．

x_{cs} サイト s を掃除する会社を表す．

y_{ca} エリア a のサイトに会社 c が割り当てられるかどうかを表す．

これら 2 つの変数の集合の関係を表すために，以下のような制約が必要です．

会社 c は，この会社がエリア a にあるサイト s に割り当てられるときだけ，エリア a に割り当てられる． すなわち，エリア a にあるサイト s について x_{cs} が 1 のときだけ，y_{ca} は 1 になる．

このとき，モデルは以下のように表されます．

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

  forward procedure cutgen
```

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

initializations from data/clparam.dat
  PARAM
end-initializations

declarations
  NSITES = PARAM(1) ! サイトの数
  NAREAS = PARAM(2) ! エリアの数
  NCONTRACTORS = PARAM(3) ! 入札する会社の数
  RA = 1..NAREAS
  RC = 1..NCONTRACTORS
  RS = 1..NSITES
  AREA: array(RS) of integer !  サイトのあるエリア
  NUMSITE: array(RA) of integer ! エリア内のサイトの数
  LOWCON: array(RA) of integer ! エリアごとの会社の数の最小値
  UPPCON: array(RA) of integer ! エリアごとの会社の数の最大値
  ADJACENT: array(RA,RA) of integer ! 1 ならサイトは隣接
  PRICE: array(RS,RC) of real ! サイトごとの会社の入札額
  x: dynamic array(RC,RS) of mpvar ! c が s にあるときだけ 1
  y: array(RC,RA) of mpvar ! 会社がエリア a のサイトに割り当てられたときだけ 1
end-declarations

initializations from data/cldata.dat
  NUMSITE
  LOWCON
  UPPCON
  ADJACENT
  PRICE
end-initializations

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

```
  forall(c in RC)
    forall(s in RS | PRICE(s,c) > 0.01)
      create(x(c,s))

  !目的：全ての掃除会社のトータルのコストを最小化
  cost := sum(c in RC, s in RS) PRICE(s,c)*x(c,s)

  !各サイトは 1 社が掃除しなければならない.
  forall(s in RS) clean(s) := sum(c in RC) x(c,s) = 1

  !同じ会社が隣接するエリアを担当できない.
  forall(c in RC, a in RA, a2 in RA | a > a2 and
         ADJACENT(a,a2) = 1)
    adj(c,a,a2) := y(c,a) + y(c,a2) <= 1

  !エリア毎の会社の数は上限と下限がある.
  forall(a in RA | LOWCON(a) > 0)
    area_low(a) := sum(c in RC) y(c,a) >= LOWCON(a)
  forall(a in RA | UPPCON(a) < NCONTRACTORS)
    area_upp(a) := sum(c in RC) y(c,a) <= UPPCON(a)

  ! a にある s について x(c,s)=1 のときだけ y(c,a)=1
  forall(c in RC, a in RA) do
    y_upp(c,a) := y(c,a) <= sum(s in RS | AREA(s)=a) x(c,s)
    y_low_area(c,a) := y(c,a) >= 1.0/NMSITE(a)*sum(s in RS |AREA(s)=a) x(c,s)
  end-do

  forall(c in RC) do
    forall(s in RS) x(c,s) is_binary
    forall(a in RA) y(c,a) is_binary
  end-do

  starttime := gettime

  cutgen

  minimize(cost) !MIP 問題を解く.
  writeln("(",gettime-starttime," secs) Global status ",
```

```
  getparam("XPRS_MIPSTATUS"),", best solution: ", getobjval)
...
```

上のモデルでは，エリア a のサイト s で x_{cs} が 1 のときはいつも y_{ca} が 1 になるように，総和のやり方を使ってインプリメントしています (この制約のつけ方は，Multiple Variable Lower Bound(MVLB) 制約と呼ばれます．)．以下の部分を変更し，より強力な定式化をすることができます．

```
forall(c in RC, a in RA)
  y_low_area(c,a) := y(c,a) >= 1.0/NMSITE(a) * sum(s in RS |
AREA(s)=a) x(c,s)
```

この部分を以下のように変更します．

```
forall(c in RC,s in RS)
  y_low_site(c,s) := y(c,AREA(s)) >= x(c,s)
```

ただし，このようにすると，制約式の数は増えます．

総計された制約は，この問題を表現するには十分ですが，定式化はあいまいで，その結果，LP 緩和では，求めたい整数解の非常に悪い近似値しか得られなくなります．そのため，大きなデータセットでは，分枝限定探索に非常に時間がかかります．これを不必要な制約を加えることなく改善するため，探索のノードの最初にカット生成ループを実装します．これは，y_low_site という違反する制約を LP 解に付け加えることによって実現します．

カット生成ループ (手続き cutgen) は以下のようにして動作します．

1. LP を解き，基底を保存する．
2. 解の値を得る．
3. 違反する制約を見つけ，これを問題に付け加える．
4. 変更された問題をロードして，前に保存した基底をロードする．

```
procedure cutgen
  declarations
    MAXCUTS=2500 ! カットの最大値
    MAXPCUTS=1000 ! パスあたりのカットの最大値
    MAXPASS=50 ! パスの最大値
    ncut,npass,npcut: integer ! カットとパスのカウンタ
    ztolrhs: real ! 0 トレランス (許容誤差範囲)
    solx: array(RC,RS) of real ! x の解の値
    soly: array(RC,RA) of real ! y の解の値
    objval, starttime: real
    cut: array(range) of linctr
  end-declarations
```

```
starttime:=gettime
setparam("XPRS_CUTSTRATEGY",0) ! 自動的なカットを禁止
setparam("XPRS_PRESOLVE",0) ! 事前解法を無効
ztolrhs:=getparam("XPRS_FEASTOL") ! 零許容範囲の値を得る
ztolrhs = ztolrhs*10
ncut:=0
npass:=0

while(ncut < MAXCUTS and npass <MAXPASS) do
  npass+=1
  npcut:=0
  minimize(XPRS_LIN, cost) ! LP を解く
  savebasis(1) ! 現在の基底を保存
  objval:=getobjval ! 目的関数の値を得る
  forall(c in RC) do ! 解の値を得る
    forall(a in RA) soly(c,a) := getsol(y(c,a))
    forall(s in RS) solx(c,s) := getsol(x(c,s))
  end-do

  ! violated な制約をみつけ, 問題に加える
  forall(s in RS)
    forall(c in RC)
      if(solx(c,s)-soly(c,AREA(s)) > ztolrhs) then
        cut(ncut) := y(c,AREA(s)) >= x(c,s)
        ncut+=1
        npcut+=1
        if(npcut>MAXCUTS or ncut > MAXCUTS) then
            break 2
        end-if
      end-if
    writeln("Pass ",npass," (",gettime-starttime," sec),
        objective value: ",objval,", cuts added: ",npcut,"
        (total: ",ncut,")")

    if(npcut=0) then
      break
    else
      loadprob(cost) ! 問題をリロード
```

```
        loadbasis(1) ! 保存されている基底をロード
      end-if
    end-do

    !カット生成の状態を表示
    write("Cut phase completed: ")
    if(ncut >= MAXCUTS) then
      writeln("space for cuts exhausted")
    elif(npass>=MAXPASS) then
      writeln("maximum number of passes reached")
    else
      writeln("No more violations")
    end-if
  end-procedure

end-model
```

A.2 列生成

列生成の手法は，問題の行列の全ての列を生成するのが実用的ではないとき，莫大な数の変数に関する線形問題を解くために使われます．非常に制限された列の集合で始まって，問題を解いた後で，列生成アルゴリズムは，現在の解を改善する 1 つまたはいくつかの列を加えます．これらの列は，(最小問題で) 負のリデューストコストを持っていなければならず，現在の解答の双対値を基に計算されます．

大きい MIP 問題を解くためには，列生成は，分枝限定探索と一緒に使われます．例題は，分枝限定探索なしの LP を順番に解いています．

例題 製紙工場は，固定した幅 $MAXWIDTH$(顧客によってオーダに基づいた更に小さいロールに切られる) の紙のロールを生産します．それらのロールは，$NWIDTHS$ の異なるサイズに切られることがあります．オーダは，幅 $i(DEMANDi)$ で要求されます．製紙工場の目的は，損失を最小限にするために，紙ロールの数をできるだけ少なくして要求を満たすことです．

ロールの総数を最小にするという目的は，現在の要求に対して，最良の切断パターンを選択することであると表現することができます．全ての可能な切断パターンをどのように計算するかが明白ではないかもしれないので，大きいロールからできる限り何度も同じ幅の小さいロールを切るという基本パターン $(PATTERN_1, \ldots, PATTERN_{NWIDTHS})$ の集合からはじめます．

切断パターン j が使われた回数を表す変数 pat_j で定義し，目的関数は，これらの変数についての総和を，全てのサイズに対する要求を満たすという条件の下で最小化することです．

```
model Papermill
  uses "mmxprs"
```

```
forward procedure colgen
forward function knapsack(c: array(range) of real,
      a: array(range) of real, b: real,
      xbest: array(range) of integer): real
forward procedure shownewpat(dj: real,
      vx: array(range) of integer)

declarations
  NWIDTHS = 5 !異なる幅の数
  RW = 1..NWIDTHS ! 幅の範囲
  RP: range ! 切断パターンの範囲
  MAXWIDTH = 94 ! ロールの最大幅
  EPS = 1e-6 ! 零許容範囲

  WIDTH: array(RW) of real ! 可能な幅
  DEMAND: array(RW) of integer ! 幅ごとの要求
  PATTERNS: array(RW,RW) of integer ! 基本パターン
  pat: array(RP) of mpvar ! パターンごとのロール
  solpat: array(RP) of real ! 変数の解
  dem: array(RW) of linctr ! 制約式
  minRolls: linctr ! 目的関数
  knap_ctr, knap_obj: linctr ! ナップザック制約
  x: array(RW) of mpvar ! ナップザック変数
end-declarations

WIDTH := [ 17, 21, 22.5, 24, 29.5]
DEMAND:= [150, 96, 48, 108, 227]

!基本パターンをつくる
forall(j in RW) PATTERNS(j,j) := floor(MAXWIDTH/WIDTH(j))

forall(j in RW) do
  create(pat(j)) ! NWIDTHS 個の変数 pat をつくる
  pat(j) is_integer ! 変数は整数で上限下限がある.
  pat(j) <= integer(ceil(DEMAND(j)/PATTERNS(j,j)))
end-do

forall(j in RW) x(j) is_integer ! ナップザック変数は整数
minRolls := sum(j in RW) pat(j) ! 目的関数
```

```
  ! 全ての要求を満たす
  forall(i in RW)
    dem(i):=sum(j in RW) PATTERNS(i,j)*pat(j) >= DEMAND(i)

  colgen ! 列生成

  minimize(minRolls) ! 新しい列を含む現在の問題の最良の整数解を計算

  writeln("Optimal solution: ",getobjval," rolls)
  write(" Rolls per pattern: ")
  forall(i in RP) write(getsol(pat(i)),", ")
  writeln
```

切断パターンの基本的なセットでは工場は要求を満たすことができますが，大きいロールを顧客のオーダよりも小さくカットしてしまって浪費し，有意な損失を負うことになるかもしれません．それゆえ，発見的に列生成を行い，より適切な切断パターンを見つけます．

以下に示した関数 colgen は，ノードのトップで実行される（M.Savelsbergh によって，これと似た切断ストック問題を解決するために提案された発見的手法である）列生成ループを定義しています．列生成ループは以下のとおりです．

1. LP を解いて基底を保存する．
2. 解の値を得る．
3. 現在の解をもとに，収益の多い切断パターンを計算する．
4. 新しい列 (切断パターン) を生成し，目的関数と対応する要求の制約に加える．
5. 変更された問題をロードして，前に保存した基底をロードする．

この問題で変数 pat_j の数を増加させることができるようにするために，これらの変数は，(添字範囲を指定せずに) 動的な配列としてプログラムの始めに宣言されます．

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

  setparam("XPRS_CUTSTRATEGY", 0) !自動的なカットを無効
  npatt:=NWIDTHS
```

```
  npass:=1

  while(true) do
    minimize(XPRS_LIN, minRolls) !LP を解く
    savebasis(1) ! 現在の基底を保存
    objval:= getobjval ! 目的関数の値を得る

    !解の値を得る
    forall(j in 1..npatt) solpat(j):=getsol(pat(j))
    forall(i in RW) dualdem(i):=getdual(dem(i))

    zbest:=knapsack(dualdem, WIDTH, MAXWIDTH, xbest) - 1

    write("Pass ", npass, ": ")
    if(zbest < EPS) then
      writeln("no profitable column found.\n")
      break
    else
      shownewpat(zbest, xbest) !新しいパターンを表示
      npatt+=1
      create(pat(npatt)) ! 新しい変数を作る
      pat(npatt) is_integer

      minRolls+= pat(npatt) ! 目的関数に新しい変数を追加
      dw:=0
      forall(i in RW) ! 制約式に新しい変数を追加
        if(xbest(i) > EPS) then
          dem(i)+= xbest(i)*pat(npatt)
          dw:= maxlist(dw, ceil(DEMAND(i)/xbest(i) ))
        end-if
      pat(npatt) <= dw ! 新しい変数の上限値をセット

      loadprob(minRolls) ! 問題をロード
      loadbasis(1) ! 保存されている基底をロード
    end-if
    npass+=1
  end-do
end-procedure
```

手続き colgen は，下のような形式の整数ナップザック問題

$$z = \max\{cx : xa \leq b, x\text{is integer}\}$$

を解くために，次のような補助的な関数 knapsack を呼び出します.

```
function knapsack(c: array(range) of real,
                  a: array(range) of real, b: real,
                  xbest: array(range) of integer): real
  ! 制約を隠す
  forall(j in RW) sethidden(dem(j),true)

  knap_ctr := sum(j in RW) a(j)*x(j) <= b
  knap_obj := sum(j in RW) c(j)*x(j)

  maximize(knap_obj) ! LP 問題を解く
  returned := getobjval ! 目的関数の値を得る

  !解の値を得る
  forall(j in RW) xbest(j) := round(getsol(x(j)))

  knap_ctr := 0 ! ナップザック制約をリセット
  knap_obj := 0 ! ナップザック目的関数をリセット
  forall(j in RW) sethidden(dem(j),false) ! 制約を隠すのをやめる
end-function
```

ナップザック問題は，メインの問題と同じモデルの中で定められている完全に独立した最適化問題です. ナップザック問題を解いているとき，オプティマイザは，ナップザック制約のみを考慮します. 関数 knapsack の最後にナップザック問題の定義は取り除かれ，メインの問題の制約が復活し，オプティマイザが関数 colgen のループの中でメインの切断ストック問題を解くために呼び出されます.

Mosel における問題の定義はインクリメンタルなので，ナップザック問題の制約はグローバルに定義されます. サブルーチンにおいて定義された制約は，その終わりで削除されません. (従って，ナップザック制約は，関数 knapsack の終わりで明示的にリセットする必要があります.)

次のような手続き colgen は，全ての新しいパターンを表示します.

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

  writeln("New pattern found with marginal cost ",dj)
  write(" Widths distribution: ")
```

```
  dw:=0
  forall(i in 1..NWIDTHS) do
    write(WIDTH(i),": ", vx(i)," "
    dw += WIDTH(i)*vx(i)
  end-do
  writeln("Total width: ",dw)
end-procedure
```

A.3 再帰

Successive Linear Programming として知られる再帰は，LP が非線形問題を解くために使うことができる手法です．LP 問題におけるいくつかの係数は，LP 変数の最適値の関数になるように定義されます．

LP 問題が解かれたあとで，係数が再評価され，LP が再び解かれます．ある仮定の下では，このプロセスは (必ずしもグローバルな最適解ではない) 局所最適解に収束します．

次のような財務計画問題を考えます．与えられた支払いのについて期末残高が 0 を達成するような年 1 回の利率 x を決定したいとします．利息は次の式に従って，年 4 回支払われます．

$$interest(t) = \left(\frac{92}{365}\right) \times balance(t) \times rate$$

時刻 $t(t = 1, \ldots, T)$ での残高は，前期 $t-1$ での残高と，支払いと利息の純益で決まります．

$$\begin{aligned} net(t) &= payments(t) - interest(t) \\ balance(t) &= balance(t-1) - net(t) \end{aligned}$$

この問題は，非線形項

$$balance(t) \times rate$$

があるため，LP だけではモデル化できません．もとの問題を LP によって近似するために，利息の変数 x を定数 X とその偏差を表す変数 dx で次のように置き換えます．

$$x = X + dx$$

すると，年 4 回の利払い $i(t)$ の式は以下のようになります．

$$\begin{aligned} i(t) &= 92/365 \times b(t-1) \times x \\ &= 92/365 \times (b(t-1) \times X + b(t-1) \times dx) \end{aligned}$$

そこで，残高 $b(t-1)$ と dx との積の項の $b(t=1)$ を，推定値 $B(t)$ と偏差 $db(t-1)$ で置き換えます．

$$i(t) = 92/365 \times (b(t-1) \times X + B(t-1) \times dx + db(t-1) \times dx)$$

これは，偏差同士の積を省略することによって以下のように近似できます.

$$i(t) \quad = \quad 92/365 \times (b(t-1) \times X + B(t-1) \times dx)$$

採算を保証するために，制約の定式化のときに，正と負の偏差のためのペナルティ変数 epl 及び emn を加えます.

$$i(t) \quad = \quad 92/365 \times (b(t-1) \times X + B(t-1) \times dx + epl - emn)$$

モデルは以下のようになります．残高変数 b(t)，偏差 dx，年 4 回の純益 n(t) は is_free を使ってフリー変数として定義します．これは，無限大からマイナス無限大までの任意の値を取りうることを意味します.

```
model fin_nlp
  uses "mmxprs"

  forward procedure solverec

  declarations
    T = 6 ! 時間の長さ
    RT = 1..T ! 時間の範囲
    P,R,V: array(RT) of real ! 支払い
    B: array(RT) of real ! 残高 b(t) の初期値
    X: real ! 利率 x の初期値

    i: array(RT) of mpvar ! 利息
    n: array(RT) of mpvar ! 純益
    b: array(RT) of mpvar ! 残高
    x: mpvar ! 利率
    dx: mpvar ! x の変化
    epl, emn: array(RT) of mpvar ! + と-の 偏差
  end-declarations

  X := 0.0
  B := [ 1, 1, 1, 1, 1, 1]
  P := [-1000, 0, 0, 0, 0, 0]
  R := [206.6, 206.6, 206.6, 206.6, 206.6, 0]
  V := [-2.95, 0, 0, 0, 0, 0]

  ! 純益 = 支払い - 利息
  forall(t in RT) net(t) := n(t) = (P(t)+R(t)+V(t)) - i(t)
```

```
  ! 期間を全体での残高
  forall(t in RT) bal(t) := b(t) = if(t>1, b(t-1), 0) - n(t)

  ! 利息の近似
  forall(t in 2..T) interest(t) := -(365/92)*i(t) +
         X*b(t-1) + B(t-1)*dx + epl(t) - emn(t) = 0

  ! 利率の決定
  def := X + dx = x

  ! 目的：利益をあげる
  feas := sum(r in RT) epl(t) + emn(t)

  i(1) := 0 ! 利息の初期値は 0
  forall(t in RT) n(t) is_free
  forall(t in 1..T-1) b(t) is_free
  b(T) := 0 ! final balance is zero
  dx is_free

  minimize(feas) ! LP を解く

  solverec ! 再帰ループ

  ! 解の表示
  writeln("\nThe interest rate is ",getsol(x))
  write(strfmt("t",5), strfmt(" ",4))
  forall(t in RT) write(strfmt("t",5), strfmt(" ",3))
  write("\nBalances ")
  forall(t in RT) write(strfmt(getsol(b(t)),8,2))
  write("\nInterest ")
  forall(t in RT) write(strfmt(getsol(i(t)),8,2))
  writeln
end-model
```

上のモデルで宣言されている手続き solverec は，まだ定義されていません．x と $b(t)(t = 1, \ldots, T-1)$ での再帰は，次のようにして実行されます．

1. 制約式 interest(t) での B(t-1) を一つ前の値 b(t-1) から求める．
2. 制約式 interest(t) での X を一つ前の値 x から求める．
3. 制約式 def での X を一つ前の値 x から求める．

アルゴリズムは，変化 dx(variation) が 0.00001(TOLERANCE) より小さくなったときに収束します.

```
procedure solverec
  declarations
    TOLERANCE = 0.000001 ! 収束条件
    variation: real ! x の変化
    BC: array(RT) of real
  end-declarations

  variation := 1.0
  ct := 0

  while(variation < TOLERANCE) do
    savebasis(1) ! 現在の\basis を保存
    ct += 1
    forall(t in 2..T)
      BC(t-1) := getsol(b(t-1)) !  b(t) の解の値を得る
    XC := getsol(x) !  x の解の値を得る
    write("Round ",ct," x:",getsol(x),"
                (variation:",variation,") ")
    writeln("Simplex iterations:",
                  getparam("XPRS_SIMPLEXITER"))
    forall(t in 2..T) do ! 係数の更新
      interest(t) += (BC(t-1) - B(t-1))*dx
      B(t-1) := BC(t-1)
      interest(t) += (XC-X)*b(t-1)
    end-do
    def+=XC-X
    X:=XC
    oldxval:=XC ! X の解の値を格納
    loadprob(feas) ! オプティマイザに問題をリロード
    loadbasis(1) ! 一つ前の\basis をリロード
    minimize(feas) ! 再び LP を解く

    variation := abs(getsol(x)-oldxval) ! 変化をみる
  end-do
end-procedure
```

X の初期値 0，B(t) の初期値 1 に対して，モデルは 5.94413%(x = 0.0594413) の利率に収束します.

A.4 目標計画法

目標計画法は，特殊な制約の集合に対する線形計画法の拡張です．目標計画法では，プリエンプティブ (lexicographic) モデルとアルキメデスモデルの 2 つのモデルが使われます．プリエンプティブモデルは，目標は優先順位に従って順序がつけられます．ある優先レベルの目標は，次のレベルにある目標よりも重要であると考えます．アルキメデスモデルでは，ターゲットに重みまたはペナルティが与えられ，重みの和を最小化します．

制約が目標を作るのに使われる場合には，目標は制約の違反を最小化することになります．制約を満たすものを見つけたときがゴールになります．

以下の例では，Mosel を用いてプリエンプティブモデルの目標計画法を実装しています．可能な限り多くの目標をみつけ，優先順位を決めます．目的は，2 つの変数 x と y について問題を解くことです．制約式は以下の通りです．

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

目標は以下の通りで，上から順に優先順位が高いとします．

$$\begin{aligned} 7x + 3y &\geq 40 \\ 10x + 5y &= 60 \\ 5x + 4y &\geq 35 \end{aligned}$$

読み易くするために，モデルは 3 つのブロックに分けられています．問題はメイン部分で述べられています．手続き preemptive はプリエンプティブ目標計画法のストラテジーで実装されています．手続き printsol はよい解を表示するものです．

```
model GoalCtr
  uses "mmxprs"

  forward procedure preemptive
  forward procedure printsol(i: integer)

  declarations
    NGOALS=3 ! 目標の数
    x,y: mpvar ! \dv
    dev: array(1..2*NGOALS) of mpvar ! 目標からのずれ
    mindev: linctr ! 目的関数
    goal: array(1..NGOALS) of linctr ! 目標の制約式
  end-declarations

  limit := 100*x + 60*y <= 600 ! 制約式

 ! ゴールの制約を定義
```

```
goal(1) := 7*x + 3*y >= 40
goal(2) := 10*x + 5*y = 60
goal(3) := 5*x + 4*y >= 35

preemptive ! プリエンプティブ目標計画法

procedure preemptive
  (!問題に順番に目標を加え，全ての目標が加えられるか
    目標が満たされるなくなるまで解く.
    目標の与えられる順番が優先順位である仮定から.
  !)

  ! 問題から目標制約を除く
  forall(i in 1..NGOALS) sethidden(goal(i),true)

  i:=0
  while(1<NGOALS) do
    i+=1
    sethidden(goal(i),false) ! 次の目標を加える
    case gettype(goal(i)) of ! s の変化を得る
      CT_GEQ: do
                deviation := dev(2*i)
                mindev += deviation
              end-do
      CT_LEQ: do
                deviation := -dev(2*i-1)
                mindex += dev(2*i-1)
              end-do
      CT_EQ: do
                deviation := dev(2*i) - dev(2*i-1)
                mindev += dev(2*i) + dev(2*i-1)
              end-do
      else writeln("Wrong constraint type")
          break
    end-case
    goal(i) += deviation
    minimize(mindev) ! LP を解く

    writeln("Solution(",i,"): x:", getsol(x),", y:",
```

```
            getsol(y))
      if(getobjval > 0) then
        writeln("Cannot satisfy goal ",i)
        break
      end-if
      goal(i) -= deviation ! remove deviation from goal
    end-do
    printsol(i) ! 解の表示
  end-procedure

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

    ATypes := [">=","<=","="]
    writeln(" Goal",strfmt("Target",12), strfmt("Value",9))
    forall(g in 1..i) writeln(strfmt(g,5),
         strfmt(ATypes(gettype(goal(g))),3),
         strfmt(-getcoeff(goal(g),9),
         strfmt(getact(goal(g)) - getsol(dev(2*g)) +
         getsol(dev(2*g-1)),9))
    forall(g in 1..NGOALS)
      if(getsol(dev(2*g)>0) then
        writeln(" Goal(",g,") deviation from target: -",
               getsol(dev(2*g)))
      elif(getsol(dev(2*g-1))>0) then
        writeln(" Goal(",g,") deviation from target: +",
               getsol(dev(2*g-1)))
      end-if
  end-procedure
end-model
```

この例では，手続き sethidden を使ってオプティマイザによって問題の定義から制約を削除することなく解いている問題から除いています．オプティマイザは効果的に一連の問題全体のサブ問題を解いています．実行すれば，最初の 2 つの目標を満足しているが，第 3 の目標は満たしていないことが分かるでしょう．

付録 B

Mosel 言語の文法ダイアグラム

B.1　主要な構造と文

（原文のまま）

B.2　式

（原文のまま）