An Introduction to PC Mizar

Ewa Bonarska

Fondation Philippe le Hodey, 1990

Translation by l'Hospitalier
2008

日本語版（5.0 校）

目　次

序（INTRODUCTION）----------4

Ⅰ．規約（CONVENTIONS）----------5

Ⅱ．アーティクルとディレクティブ（ARTICLE AND DIRECTIVES）--------7

Ⅱ.1. ボキャブラリ指令----------8

Ⅱ.2. 識別子----------11

Ⅱ.3. Hidden ボキャブラリ----------14

Ⅱ.4. データベース指令----------15

Ⅱ.5. Mizarの予約語、予約シンボル、数----------20

Ⅲ．項と式（TERMS AND FORMULAS）----------23

Ⅲ.1. 項（Terms）----------23

Ⅲ.2. 型（Types）----------29

Ⅲ.3. 変数の予約----------33

Ⅲ.4. 原子式（Atomic formulas）----------35

Ⅲ.5. 命題結合により原子式から形成された式----------37

Ⅲ.6. 量化式（Quantified formulas）----------38

Ⅲ.7. Semantic correlates----------49

Ⅳ．Mizarでの文の証明（PROVING SENTENCES IN MIZAR）----------53

Ⅳ.1. Justifications----------53

Ⅳ.2. 証明のスケルトン----------64

Ⅳ.2.1. 直接証明----------65

Ⅳ.2.2. 間接証明----------84

Ⅳ. 2. 3. 新しいMizarの構文について----------91

Ⅳ.3. その他の Mizar 構文----------93

Ⅳ. 3. 1 反復等号（ITERATIVE EQUALITY）----------93

Ⅳ. 3. 2 DIFFUSE STATEMENT----------97

Ⅳ. 3. 3 型変更のステートメント----------103

文献(BIBLIOGRAPHY)----------107

索引(INDEX) ----------108

Roman Matuszewski

---

dr Roman Matuszewski, University of Białystok, Poland
http://mizar.org/people/romat/

警　告

このマニュアルは原著成立が 1991 年なので、現行の PC Mizar システムに適用できない部分があります。　読者各位は最新の Mizar システム（March、2008 現在で Version7.8.10, MML Version: 4.100.1011）についての情報を Mizar Home Page http://mizar.org/ などから入手してください。

序（INTRODUCTION）

このテキストは PC Mizar システムの基本構成の記述を含みますが、完全にではありません。　テキストは４章からなりアネックス（付録）は多くの例を含みます　（訳注：アネックスは欠落しています)。

第Ｉ章は術語の問題と記号化（symbolism）を議論します。　第Ⅱ章では Mizar の基となる構造、アーティクルとディレクティブについて述べます。　識別子（identifiers)、予約語、シンボル、数（numerals）も議論します。　第Ⅲ章では式に関することを、第Ⅳ章では定理の証明に関することを述べます。

テキストは主に Mizar における統語論（syntactics、訳注：言語要素を article にまとめること、構文論）に関したものです。　証明規則の説明に不可欠な意味論（semantics）の要素については Ⅲ.7 “Semantic correlates” で議論します。

テキストには本文と付録の両方に多数の例があります（主に一般位相空間についてです)。　これは Mizar の学習や、一人で Mizar 言語のアーティクルを書く事の両方に役立つと思います。

著者は Dr. A. Trybulec と G. Bancerek の貴重な提案とコメントに多くを負っています。　このテキストを書くのにとても役に立ちました。

PC Mizar システムは A. Trybulec と Cz. Byliński により実装されました。Andrzej Trybulec は Mizar 言語の作者です。

Ⅰ．規約（CONVENTIONS）

すべての Mizar アーティクルは、一連の ASCII シンボル（ある種の順番に配列された固定の記号コード、 訳注：American Standard Code for Information Interchange）で、コントロール・コード（control signs）、記号 No.127 と No.255 以外、から構成されます。 しかし、このテキストに示す Mizar アーティクルの断片は ASCII コードにない（𝔉 𝔊 𝔰 のような）記号を含みます。 それらの記号はテキストの読みやすさを向上させるために使用されます。

以下のテーブルは Mizar アーティクルで使用を許されていないにもかかわらず本文で使用されたシンボルと、その標準 ASCII に類似する文字のリストです。

さらにテキストは:

list-…

という形の記銘（inscriptions）を含み、これはハイフン ”-“ でリンクされた語からなる記銘と同様にリスト（lists）という術語で呼ばれます。 例えば、

segment-of-qualified-variables,
symbol-of-functor.

などです。 ハイフネーションは、ある種の（訳注：語句の）全体はどのように語を結合して作成したのかを示そうとしています。 さらに、ある種の語は**太字（bold type）**で書くことにします。 それらは Mizar のために予約された語であり、その意味は Mizar 言語での定義により厳格に決定されています。 この印字上の区別は読者の注意を引き、少なくとも記憶しやすくします。 このテキストで全ての予約語とシンボルを見つけることができるので、注意して読んでください。

Symbol in this book	Representation in ASCII
$\mathcal{F}$	F
$\mathcal{G}$	G
$\mathcal{H}$	H
$\subseteq$	c=
^{c}	‘
$\cup$	U
$\emptyset$	[234]
Ω	[237]
$\in$	[238]
$\cap$	[239]
$\leq$	[243]
$\neq$	<>

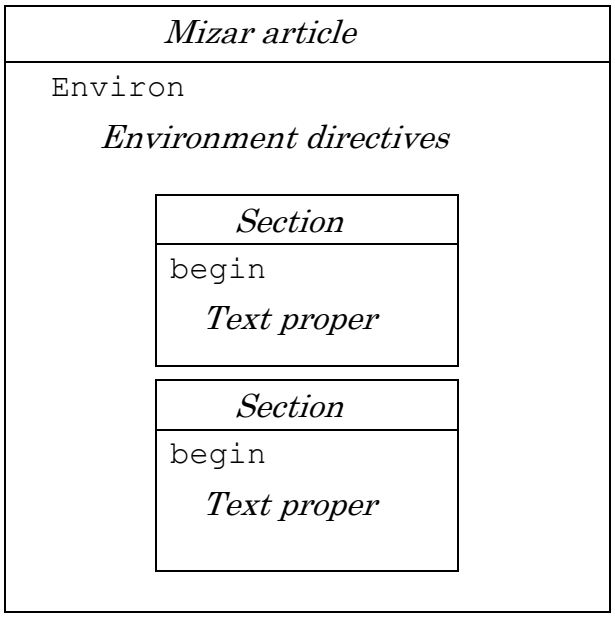
つぎのシンボルが採用されています。

$\mathcal{G}$ - 位相空間、
$\mathcal{H}$ - 位相空間 $\mathcal{G}$ の部分空間、
$\mathcal{F}$ - 位相空間 $\mathcal{G}$ の部分集合族、
A,B,G,P,Q - 位相空間 $\mathcal{G}$ の部分集合、
X,Y,M,N - 集合、
p, q - 位相空間 $\mathcal{G}$ の点、
k,l,n - 自然数、
x - 任意のオブジェクト。

Ⅱ．アーティクルとディレクティブ（ARTICLE AND DIRECTIVES）

Mizarテキストのファイルは*Mizar article*という術語で呼ばれます。 ファイルの名前は記号８個を超えない：文字、数字（figures）、アンダースコア（_）、そしてアポストロフィ記号（'）で構成されます。 数（numeral）だけや、Mizarの予約語の使用は許されません。 さらにファイル名は `.miz` の拡張子を持たないといけません。

いくつかファイルの名前の例： `W1.miz`, `''_'.miz`, `x.miz`を挙げてみます。 校正（proof）の明解さの観点から`''_'`や、それ以外であっても同程度に読みにくい記銘法をファイルの名前として使うのは推奨できません。 *Mizar アーティクル*は：*環境指令（environment directives）*と、**begin**という語でお互いに区切られた*セクション（sections）*の一連の続き、の２つの部分から構成されます。 環境の指令には**environ**という語が先行しなければなりませんが、これは全ての*Mizar article*で、その語のある場所が開始点（open）であることを意味します。



*テキスト本文（text proper）*は、定理の証明、正当な条件に関する定義、スキーム（schemata）の証明などを含みます。 空でない正しい本文（non-empty *text proper*）を書くためには、アーティクルの著者が*environment directives*により、方向づけられる（can be organized）ような環境を必要とします。 それら（訳

注：の環境）はテキスト本文（*text proper*）のMizarによる正しい読み込みと解釈のために不可欠な情報の項目を含み、証明の基盤となります。　より厳密には、*environment directives*はデータベースを参照し、アーティクルの中で現行のライブラリのどの要素が使われるかを指示します。

environment directives は：

vocabulary α;
signature β;
definitions γ;
theorems δ;
schemes ε;

を含みます。　ここで

α - ボキャブラリ・ファイル名（例えば、TOPCON, ANAL）、
β - シグネチャ・ファイル名（例えば、TOPS_1, PRE_TOPIC, SUBSET_1）、
γ - 定義ファイル名（例えば、TARSKI, BOOLE）、
δ - 定理ファイル名（例えば、CONNSP_1, REAL_1）、
ε - スキーム・ファイル名（例えば、NAT_1）

です。

II.1.　ボキャブラリ指令（Vocabulary directive）

指令

vocabulary α;

には*ボキャブラリ指令*という術語が使われ、*ボキャブラリ指令*以外は*データベース指令*という術語が使用されます。　全ての*Mizar*アーティクルは、複数のシンボルから構成されます。　それらのいくつかは自動的に導入されますが（hidden symbols）、それ以外は*ボキャブラリ指令*での参照により導入されます。　というわけでボキャブラリ指令は必須です。　ボキャブラリは .voc 拡張子を持つ単一のファイルで構成されます。 そのファイルは修飾子(qualifiers)を持つシンボルのリスト、およびfunctorシンボルの拘束力の強さの指定を保持します。　例えばボキャブラリを形成するTOPCON.vocファイルは以下のようになります。

TOPCON.voc
OCl 128
OFr 128
Oskl 128
Ucarrier
Utopology
GTopStruct
Ris_open
Ris_closed
Ris_open_closed
Rare_joined
Ris_a_component_of
Ris_a_cover_of
MTopSpace
MPoint
MSubSpace
Mmap

一番左のカラム（列）には修飾子（qualifier）があります。　その次のカラムからスペースまではシンボルです。　ある意味で修飾子はシンボルの性格を決めます。　例えば修飾子 O は、それに続くシンボルがfunctorのシンボルであることを意味します。　一方、修飾子 R はそれに続くシンボルは述語のシンボルであることを意味します。　したがってシンボル

Cl, Fr, skl

は functor のシンボルです。　一方

is_open, is_closed, is_open_closed, is_continuous,
are_joined, is_a_component_of, is_a_cover_of

は述語のシンボルです。

シンボル　Cl, Fr, skl はそれぞれ：閉包の、位相空間の部分集合の境界の、位相空間の点の構成要素（component）の演算（operation）を示します。

シンボル is_open, is_closed, is_open_closed は位相空間の部分集合のために定義された述語として使われ、それぞれ集合が開集合、閉集合、開集合であると同時に閉集合（open-closed）であることを示します。　シンボル is_continuous は位相空間の連続写像の特性（property）を示すのに使われ

ます。　シンボル are_separated は位相空間の部分集合の間の関係を示し、それらは一つの同じ構成要素（component）に属すると言っています。　シンボル is_a_component_of は２つの述語を意味し：一つは位相空間のその部分集合は、その位相空間における最大のコンパクト集合（構成要素）であると言い、他の一つは位相空間のもう一つの部分集合の構成要素であると言っています。　シンボル is_a_cover_of は位相空間の被覆であるという特性（property）を示します。

それ以外の TOPCON.voc ファイルに出現する修飾子は：

G - 構造体（structure）のシンボルの修飾子、
U - 選択子（selector）のシンボルの修飾子、
M - モード（mode）のシンボルの修飾子

です。 TopStruct というシンボルは位相空間の構造体の意味で使用され、シンボル topology と carrier は、それぞれトポロジー（位相空間）とその台集合を意味します。　モードのシンボルの TopSpace, SubSpace, Point, map という記銘はそれぞれ位相空間、部分位相空間、位相空間の点、位相空間の間の写像の意味で使われます。

functorのカッコの修飾子（qualifier）は

K - functor の左カッコ、
L - functor の右カッコ

となります。　スキーム（schema）の識別子はボキャブラリには導入されません。　絶対値の定義に使われる functor のカッコのシンボルを保持している ANAL.voc ファイルがあります。

ANAL.voc
K |.
L .|
Osgn

記銘：

<*,*>, <:,:>

は functor のカッコの別の例です。　これらはそれぞれ、有限の数列、関数の対の関数を意味します。　シンボルが HIDDEN.voc ファイルにあるfunctorのカッコのペアもあります。　**HIDDEN.voc ファイルはすべてのアーティクルに自動的に結合されます。**

さらに、ボキャブラリ・ファイルは拘束力の強さ、あるいは優先度を指示します。　これは functor のシンボルに限って適用されます。　ある functor の拘束力の強さはシンボルに続く数により指定されます。

注釈（Remark）：　ある functor の優先度を設定する数字は少なくとも一つのスペースでその functor のシンボルから分離されている必要があります。修飾子（qualifier）とそれに対応するボキャブラリ・シンボルにはスペースが無くてもかまいません。

自明なことですが、 functor のシンボルは数字が大きいとより強く結合します。　functor の優先度は区間 <0;255> の自然数で設定されます。　ボキャブラリ Topcon のすべての functor のシンボルは優先度128を持ちます。　ある種の functor のシンボルは優先度を設定する数字がありませんが、これは優先度を持たないのではありません。　その優先度は与えられていて、数値は64です。　これが標準の優先度となります。

例えば、上記のボキャブラリ Anal で見られる functor sgn のシンボルの拘束力は与えられていません。

注釈：　述語シンボルの拘束力は、常に functor のシンボルよりも弱いので、与えられません。

functor *のシンボルの拘束力*の概念は式の演算の実行順序（sequence）とリンクしています。　例えば、２つの式：

$$\mathrm{Cl}\ \ P^{c} \quad と \quad \Omega\, G \cap Q$$

を考えて見ましょう。　シンボル ᶜ の拘束力はシンボル Cl のそれ（128）より大きいので（150に達します） Cl P^{c} という記銘は集合 P の補集合の閉包とみなされます。　つまり、Cl (P^{c}) という記銘と同じです。　それは２番目のケースでも同様です。　シンボル Ω の優先度は128で、シンボル ∩ のそれは標準の64です。　ですから記銘 Ω G ∩ Q は同じく (Ω G) ∩ Q と解釈されます。　ある種のシンボルの優先度の知識は、少なくともアーティクルの余計なカッコの使用を回避するのに有効です。

Ⅱ.2. 識別子（Identifiers）

ASCII のコントロール・コード（0 から 31 までの序数の記号）、スペース（ASCII code 32）、そしてアスキー・コードで 127 から 255 までの記号を含む記銘はボキャブラリ・シンボルには使えません。

Mizar article には識別子（identifiers）という術語の記銘（inscription）が

あります。
識別子に関する記銘はどういう性質をもつのでしょうか？　*識別子 (identifiers)* というのは、任意の、空でない（non-empty）、一続きの記号列です。　記号は文字、数字、アンダースコア（_）、そしてアポストロフィ（′）ですが、Mizar の予約語、予約シンボルや数（numerals）であってはなりません（Ⅱ．5 を見て下さい）。　識別子の記銘の記号の長さは 16 を超えてはなりません。　さもないとその識別子の記銘はボキャブラリ・シンボルであるかも知れませんから。しかし逆は成り立ちません。

識別子は：

a) プライベート functor と述語、
b) 変数、
c) ラベル。

を表現するために使用されます。
そこで変数の識別子、プライベート functor と述語の識別子について話をしましょう（もしプライベート functor や述語でなければシンボルの話ということになります）。

例として、annex にある `Z1.lst` ファイルの中の識別子について述べて（specify）見ましょう。（：欠落）　それらは：

- 変数の識別子：
  `T, P,`
- ラベルの識別子：
  `Z1, Z2.`

です。　ラベルの識別子は参照（レファランス）の例です。　レファランスは以前にすでに仮定されたか（assumed）、あるいは立証された（substantiated）文（sentences）の参照を可能にします。

参照
- ローカル － *ラベルの識別子*
- ライブラリ － *ファイル・シンボル*：
  - 数字（number）
  - `def` 数字（number）

ローカル参照の例を上に示します。

ライブラリ参照は記銘：

TOPS_1:28, BOOLE:1, TARSKI:4, REAL_1:5, SUBSET_1:14, PRE_TOPC:34 などがその例（exemplified）です。

ライブラリ参照は確立した（definite）Mizar ライブラリ定理への参照になります。 例えばライブラリ参照 TOPS_1:28 はファイル TOPS_1.abs に記録されている No.28 の定理の参照になります。 このライブラリ参照に対し、作成中のアーティクルの文に適用されるローカル参照は、自由に文に割り当てることができます。

文にはラベルを割り当てることができるので、テキストの後のほうでその文を参照することができます。 signature 指令の間で起きるように（**Ⅲ.1** を見てください)、ラベルの識別子の間のオーバーライドという現象も起きるかもしれません。

ファイル名の識別子の判別方法（DISCRIMINANTS）

- ファイル名の識別子は最大で 8 個の記号で構成されます。
- ファイル識別子は：文字、数字、アンダースコア（_）、そしてアポストロフィ（'）で作成されます。
- ファイル識別子では大文字と小文字は同一視されます。例えば row, Row, そして ROW は一つの同じファイル名となります。

採用された規約（convention）では、ファイル名は常に大文字で書くことになっています。

識別子の記銘は、環境部で使われたボキャブラリ・ファイルと密接な関連を持ちます。 重要な点はそれらのファイルにあるシンボルは識別子にはなれないということです。 付録（annex：欠落）にある例 Z2.lst でレポートされたエラーを忘れてはいけません。 それは変数の識別子として Fr という記銘を用いた結果として起こりました。 Fr はボキャブラリ Topcon に含まれる functor のシンボルであり、そのボキャブラリは環境部に結合されることに注意して下さい。 というわけで、前に言ったように記銘 Fr は変数の識別子として使うことは許されません。

注釈：アーティクルを書く者には識別子の構築について広範な自由があります。識別子として機能する記銘はできる限り可読性を良くすべきだという事実に注意を喚起するのはこういう（自由があるという）理由からです。 それはアーティクルの明解さと審美的な外見に貢献するからです。

Ⅱ.3. Hidden ボキャブラリ（Hidden vocabulary）

HIDDEN.voc
MAny
MElement
MDOMAIN
MSubset
MSET_DOMAIN
MSUBDOMAIN
MReal
MNat
K[:
L:]
Obool 128
OREAL 255
ONAT 255
O+ 32
O・
R<>
R∈
R≤
R≥
R<
R>

functor のシンボル ＋ と ・ はそれぞれ項の加算と乗算を定義するために使用され、その型は Element of REAL にまで拡張されます。 シンボル ∈ と ≤ はそれぞれ、集合とその要素の関係、集合の順序関係を示すために使用されます。 ∈ でシンボライズされた述語の正しい使用法は、左側の引数の型が Any に拡張され得ること、右側の引数のそれは set まで拡張され得るということから構成されます。 ≤ でシンボライズされる述語はその型が Element of Real まで拡張可能であるオブジェクトと定義されます。 Bool という記銘はある集合の全ての部分集合の集合族を表す functor のシンボルです。
Hidden ボキャブラリに見られる functor のカッコのシンボルは集合の直積（Cartesian product）を表すものとして採用されました。 そのボキャブラリの残りのシンボルについては型のためのセクションで議論しましょう。

Ⅱ.4. データベース指令（Data base directives）

はじめに *signature 指令*の議論をしましょう。　指令

signature β**;**

は自動的にファイル、β.sgn, β.nfr, そして β.typ を一つに結合します。これらはボキャブラリで導入されたシンボルがどのように使用されるかについての情報を持っています。　例えばファイル β.sgn は定義されたオブジェクトの引数の型と定義のパターンをリストアップします。　β.nfr は定義されたオブジェクト（functor、述語、モード）の書式（formats）の記述を保持します。ここではスキーム（schemata）のための書式は与えられません。　書式は引数の数についての情報を提供します。　同一のシンボルが複数の書式をもつことがあります。　例えば、シンボル Ø はアーティクル BOOLE の中で空集合を表すために 0-0 引数（ゼロ左側引数 - ゼロ右側引数、テーブルⅢ.１を見てください）で使用されます。　一方アーティクル PRE_TOPC では書式 0-1 引数（ゼロ左側引数 - １右側引数、テーブルⅡ.１を見て下さい）で、位相空間の開集合の族の最小の要素を表します。　両方のケースとも functor のシンボルに属するので優先度は同じです。　β.typ の内容は functor の結果の型とモードの拡張の型を保持します。

例えば、環境部に

signature BOOL;

指令を付け加えると結果としてシンボル ∩（ボキャブラリ Boole －ファイル BOOLE.voc にあります）を加えたことになり、これは共通集合を作る２項演算（two-argument operation）を意味し、それを適正に使えるようになります。左側引数と右側引数はいずれも集合です。　さらに演算 ∩ の結果もまた集合です。　ここでのシンボル ∩ が解釈される文脈は intersection の定義に従い、それはアーティクル Boole で与えられます(ここで X,Y は集合の識別子です)。

```
definition
    let X,Y;
    func X ∩ Y -> set means
    x ∈ it iff x ∈ X & x ∈ Y;
end;
```

定義の一部の X ∩ Y -> set を分析してみましょう(//??use->us)。　それは X ∩ Y の演算 ∩ は２項演算であるということです（集合 X と Y は引数です）。　シンボル -> の後のシンボル set は演算 ∩ の結果が一つの集合で

あることを知らせます。

指令

```
signature  BOOL;
```

はアーティクル Boole 内の(//??everriden->overridden)（オーバーライドされていない）すべての定義へのアクセスを可能にします。　これは、シンボル ∩ が表す共通集合（intersection）の演算の定義に適用されます（引数の数、引数の型、演算結果の型）。

しかしシンボル ∩ で表される演算は異なった解釈をされるかもしれません。アーティクル PRE_TOPC はシンボル ∩ の再定義を含みます：

```
definition
    let G, P, Q;
    redefine
    func P ∩ Q -> Subset of G;
end;
```

ここで P,Q は位相空間 G の部分集合の変数の識別子です。
指令

```
signature  PRE_TOPC;
```

が環境部に付加されると、シンボル ∩ は左右両側の引数がいずれも位相空間 G の部分集合である場合の共通集合を求める２項演算を表すのに使用されます。さらに、∩ 演算の結果もまた位相空間 G の部分集合です。

その場合 signature指令の適用を環境部に含ませるべきなのでしょうか？シンボルΩ、′、∩、∪、\（訳注：以後 \（バックスラッシュ）は日本語フォントの ¥ で表します）で表される位相空間 G の部分集合に対する演算はアーティクル PRE_TOPC で定義されています。　最初の２つのシンボルの場合は定義を、残りについては再定義をしなければなりません。　上のシンボルで表される演算の引数である変数の識別子は、練習問題では、位相空間 G の部分集合として予約されているので、指令

```
signature  PRE_TOPC;
```

を環境部に付加せねばなりません。　またモード TopSpace に関する情報もあるでしょう。　シンボル Subset を持つモードはボキャブラリ HIDDEN のなかにあるはずです。　これは全てのアーティクルに自動的に付加されるので、環境部の指令のなかに含めることはできません。　さらに、そのなかのシンボルの使用情報は該当するsignature指令無しでPC Mizarの言語プロセッサで自

動的に使用されます。

annexの例 Z4.lst と Z5.lst は適切なsignature指令が無いためのエラーとなります。(: 欠落)

注釈： signature指令が指定（specify）される順序が重要であることがあります。 それは同一のシンボルの再定義のケースです。 有効な再定義は常に環境部で指定された最後のsignatureのものです。 順序が誤っているときは、定義されたオブジェクトはオーバーライド(///??everriden->overridden)されてしまうでしょう。

下の例は **signature** PRE_TOPC; で定義された集合の共通集合を求める演算のオーバーライディングを示します。 エラー No.103 が報告された個所を示します。

```
environ
    vocabulary SUB_OP;
    vocabulary BOOLE;
    vocabulary TOPCON;
    signature PRE_TOPC;
    signature BOOLE;
    theorem BOOLE;
    theorem TOPS_1;
begin
  reserve 𝒢 for TopSpace, P, Q, for Subset of 𝒢;
  (P ∩ Q)^c = P^c ∪ Q^c
        *103
    proof
     (P ∩ Q)^c = Ω𝒢 ¥ (P ∩ Q) by TOPS_1:5
          *103
        .= (Ω 𝒢 ¥ P) ∪ (Ω 𝒢 ¥ Q) by BOOLE:86
        .= P^c ∪(Ω𝒢 ¥ Q) by TOPS_1:5
        .= P^c ∪ Q^c by TOPS_1:5;
    hence thesis;
  end;
```

（annex の例 Z6 を考えてみてください。 : 欠落）

最後のsignature指令が指令 **signature** BOOLE; なので、シンボル∩で表される演算はアーティクル BOOLE（位相空間の部分集合もまた集合です）の中

で定義された意味で使われます。　その定義に従って、共通部分（論理積）をとる演算の結果も集合です。　こうして共通部分P ∩ Qは集合となります。　しかし閉包演算（closure operation）は fixed set（訳注：型の設定された集合？）の部分集合についてのみ定義されています。　それが表現(P ∩ Q)^c の後にエラーの指示がある理由です。

今考えている例の中で出現する signature の順序を変えることで指令 signature PRE_TOPC; のオーバーライディングを避けることができます（例 Z7.lst でやったように：欠落）。

証明は時には、定義の拡張（definitional expansion）という方法で行われます。　そういうケースでは指令 **definitions** γ; を環境（environ）に加えるべきです。　定義の拡張による証明は例の中で示します。　定理の証明は以下のように与えられます：

任意の集合 X,Y について：　X ∩ Y ⊆ Y 。

証明は（Mizar記法ではありません！）以下のようになります：

a は任意で（arbitrary）ただし fixed（型の設定されている）の、
a ∈ X∩Y であるものとします。（訳注：a と a の使いわけは原文のまま）

1）　a ∈ X ∩ Y　（証明の前提となる仮定）
2）　a ∈ X ∧ a ∈ Y　（1, 集合の共通部分（論理積）の定義）
3）　a ∈ Y　（2, 論理和（conjunction）の切捨て（omission）の法則）

それは a の選択の恣意性（arbitrariness）と X ⊆ Y という定義の拡張から得られます（follows from）。

X ⊆ Y ⇔ ∀a（a ∈ X ⇒ a ∈ Y）- 包含（inclusion）の定義の拡張。

Mizar で定義の拡張を参照して上記の定理を証明するときには、環境に指令

definitions TARSKI;

を追加しなければなりません。　なぜならアーティクル TARSKI には包含の定義があり、それは以下のようになります：

pred X ⊆ Y **means** x ∈ X **implies** x ∈ Y;

そして集合のqualityの再定義があり、これはアーティクル BOOLE にあって以下のようになっています：

pred X = Y **means** X ⊆ Y & Y ⊆ X;

ファイル art.lst のなかの例の一つでは（: 欠落）、定理は２つの方法で証明されています。　両方のケースで包含（inclusion）の定義の拡張と集合の同等性（equality）の定義の拡張が使用されています。　これが*環境指令に2つの定義指令*:　**definitions** TARSKI; と **definitions** BOOLE; がある理由です。

定義指令　　**definitions** γ;

はファイル γ.def を自動的に付加し、それは定義項（//??definitienses-> definienses）を含みます。　（definiens － functor、 述語、モード、属性（//??a->an）の定義の中に現れる語 means の後に続く表現を指します）

定理指令　　**theorem** δ;

はファイル δ.miz の中の定理を使用可能にします。　その指令を書くことはそのアーティクルの中の定理の内容を含むファイル δ.the の自動的結合という結果をもたらします。

指令　　**schemes** ε;

はファイル ε.miz にあるスキーム（schemata）の内容を収納しているファイル ε.sch を自動的結合して、そのファイルのスキームを使用可能にします。
例えば、帰納法のスキーム（induction schema）はアーティクル NAT_1 にあります。　それを使うために指令 **schemes** NAT_1; を環境部に付加せねばなりません。 それは語 **environ** と語 **begin** の間にその指令を挿入することです。
テキストが数個のボキャブラリを要求する場合は指令

vocabulary *name-of-file*;

を対応するボキャブラリ・ファイルの名前の数だけ繰り返さなければなりません。　これ以外の指令の場合については類推しながらやってみて下さい。

注釈: 同一のファイル名の指令を繰り返した場合エラーとなります。　しかし、例 Z7.lst のなかの例のように（: 欠落）、与えられたファイルにとって余計な指令が加えられた場合にはエラーにはなりませんが、無効な指令になります。ここでは **signature** BOOLE; *が余計な指令です。*

Mizar のデータベース機構（organization）についての短い記述

Mizar のメインディレクトリ ¥MIZAR には２つサブディレクトリがあります。

¥DICT – ボキャブラリ・ファイルを対象とします
（拡張子 .voc を持つファイル）

¥PREL – LIBRARIAN と呼ぶプログラムで作成されたライブラリ・ファイルを対象とします。 それらのファイルは自動的に作成され、拡張子：
.sgn, .nfr, .typ, .def, .the, .sch
を持ちます。 これらはデータベースを構成します。
コンピュータのディスク上のこれらのサブディレクトリの存在は Mizar プロセッサが Mizar article の処理を可能にするのに必要な情報を引き出すために必要です。 ¥ABSTR サブディレクトリはしばしば付加的に作成されます。

¥ABSTR – 特別な処理の後で Mizar プロセッサが Mizar article から獲得したライブラリ・ファイルを対象とします。 このサブディレクトリのファイルは*アブストラクト（abstract）*と呼ばれ拡張子 .abs を持ちます。 アブストラクトはその主要部分に、定理、定義、スキームの内容を保持します。 証明は含みません。 ファイル #.abs 中の定理（ここで # はファイルの名前を表します）は番号を持ちます。 #.abs のすべての定理には：

:: # : *number-of-theorem*

という形式のヘッドラインが先行します。

サブディレクトリ ¥ABSTR はユーザーにとって補助的な役割しかありません。 そのサブディレクトリのファイルを精読（perusing）すれば、どの定理が既に Mizar で証明されているかを知ることができます。 さらには、あなたが望めば、ある種の文の証明において、*Main Mizar Library* のファイルを参照することができます。 それからファイルの名前と定理の番号を読み取り、正しい場所でそれを参照することができます。 しかしサブディレクトリ ¥ABSTR がコンピュータの記憶装置に記録されている必要はありません。 Mizar の言語プロセッサはサブディレクトリ ¥DICT と ¥PREL のファイルから与えられる情報だけを使用します。

Ⅱ．5． Mizarの予約語、予約シンボル、数（Words reserved for Mizar. Reserved symbols. Numerals）

Mizar の予約語は英語から抽出されています。 それらは Mizar 言語の定義によって意味が規定されている記銘です。 例えば **environ** は Mizar で予約されている語です。 全ての Mizar article においてそれが開始点となりま

す。　その語はアーティクルの中で最初に一回だけ現れます。　他の文脈での **environ** の使用はエラーになります。　他の予約語もまた精密に（precisely）定義された意味をもちます。　ここで付け加えますが Mizar のために予約されているシンボルもあります。　それらの意味も前もって決定されています。　それらには：

```
=    &    ,    ;    :    (    )    [    ]    {    }    ->
.=   <>   (#   #)
$1   $2   $3   $4   $5   $6   $7   $8
```

などがあります。　数（numerals）はゼロ（0）を含む任意の有限の数字（figures）の続きであって、0 で始まることはありません。　Mizar の言語プロセッサは <0;255> の間の数を取り扱うことができます。　例えば、00,0103 は数（numerals）ではありません。

Mizar のための予約語のリスト

aggregate	**and**	**antonym**
as	**associativity**	**assume**
attr	**be**	**begin**
being	**by**	**canceled**
coherence	**compatibility**	**consider**
consistency	**contradiction**	**correctness**
def	**define**	**definition**
definitions	**end**	**environ**
ex	**exactly**	**existence**
for	**from**	**func**
given	**hence**	**hereby**
holds	**if**	**iff**
implies	**irreflexivity**	**is**
it	**let**	**means**
mode	**non**	**not**
now	**of**	**or**
otherwise	**over**	**per**
pred	**prefix**	**proof**
provided	**qua**	**reconsider**
redefine	**reflexivity**	**reserve**
scheme	**schemes**	**selector**

set	signature	st
struct	such	symmetry
synonym	take	that
the	then	theorem
theorems	thesis	thus
uniqueness	vocabulary	where

Ⅲ. 項と式（TERMS AND FORMULAS）

Mizar アーティクルを正しく書くためには、Mizar 文を構成する技術が必要条件です。 それがこの章で Mizar 文に関連する基本的要素について議論する理由です。 それは項と式です。 まず項からはじめましょう。

Ⅲ.１. 項（Terms）

項の集合は以下の条件を満たす最小の集合です：

a) 変数と定数は項です。
b) もし $t_1, \ldots, t_p$ が項ならば F は p 個の引数を持つ functor のシンボルであり、$F(t_1, \ldots, t_p)$ は項です。

しかし Mizar 言語では、項の概念はより広く解釈されます。 Mizar の項とはあるカテゴリーのもとで次にリストアップするような記銘のことです。

(1). 変数の識別子は項です。
例えば： `P, TS, q` というような記銘がそうです：

(2). 数（NUMERALS）は項です。
例えば: `2, 178, 77.`

(3). 下記の形の表現は項です。
list-of-leftside-arguments symbol-of-functor list-of-rightside-arguments

functor シンボルはボキャブラリに存在していないといけません。 ボキャブラリ内の記銘はシンボルと呼ばれます。 これは functor シンボルや述語等に言及するとき、それらがボキャブラリ内の functor や述語のシンボル等を意味する理由となります。

引数リストの引数の数（右側と左側の両方です）はゼロでもよいのです。 それは functor Ø の場合です。 さらに左側引数がゼロの場合や、右側引数がゼロのケースがあると思います。 例を下に載せます。

位相空間に関するアーティクルで使用された以下の functor シンボル：

`Ø、∪、∩、‘、Cl、 Int、 Fr`

や、ここにないシンボルを考えてみてください。 下のテーブルはそれらの項の引数の数を示します。

Functor Symbol	Term	Number of left-side arguments	Number of right-side arguments
Ø	Ø	0	0
Ø	Ø Ꞛ	0	1
∪	P ∪ Q	1	1
∩	P ∩ Q	1	1
ᶜ	P ᶜ	1	0
Int	Int P	0	1
Cl	Cl P	0	1
Fr	Fr P	0	1
FinUnion	FinUnion(B,f)	0	2
PLANE	PLANE(A,B,C)	0	3
All	All(x,y,z,H)	0	4
·	o.(a,b)	1	2
*	D*	1	0

もし（左側と右側の両方の）引数のリストが２つ以上の引数から構成されていれば、引数のリストはカッコ（ と ）の中に置かなければなりません。 ちょうど上のテーブルの該当する項で行われているように。

(4) . *leftside-functor-bracket non-zero-list-of-terms rightside-functor-bracket* という表現は項です。

list-of-terms はコンマで区切られた項の有限の列です。

例：

- 集合の直積（Cartesian product）：
  `[:M1, M2:]` 、 `[:M1,M2,M3:]` 、 `[:M1,M2,M3,M4:]`
- aとa-bの絶対値：
  `|.a.|` 、 `|.a - b.|`
- それぞれ、1 あるいは 2 の長さの有限の数列：
  `<*k*>` 、 `<*k,l*>`
- 関数 f と g のペアの関数：
  `<:f,g:>`

アーティクルの作者は既にそのための新しいシンボルをボキャブラリに導入しているかも知れないので、それらのすべてが functor のカッコを使うというわけではありません。

注釈：functor のカッコはペアで使わねばなりません。　全てのペアで（訳注：ペアの中では）、同じ種類のカッコでないといけません。
例えばある表現で記銘［が functor の左側カッコとして使用されたら、記銘］がその表現の中の functor の右側カッコでないとなりません。

その間に項が無い functor のカッコのペアは項ではありません。　特別な種類の functor として扱われる２つの型のカッコがあります。　それらは：
[、] と { 、 } です。
これらは次の形の項を構築するのに使うことができます：
[*list-of terms*] あるいは { *list-of-terms* } 。

例：

- 順序対、３対子、そして４対子
  [x,y] 、 [x,y,z] 、 [x,y,z,v]
- シングルトン {x} 、ペア {x,y} 、さらに８要素までの有限の集合。
  {x1, x2, x3}
  {x1, x2, x3, x4}
  {x1, x2, x3, x4, x5}
  {x1, x2, x3, x4, x5, x6}
  {x1, x2, x3, x4, x5, x6, x7}
  {x1, x2, x3, x4, x5, x6, x7, x8} .

(5) . { *term* : *formula* } という形の記銘は項です。
この種の項は *Frænkel's operators* と呼ばれます。　次の表現を例として引用しましょう：
{ x： x ≤ 8 } 、
ここで x は変数の識別子で、Real 型であると予約されています。
式は後で議論します（次のセクションを見て下さい）。
今、この項に現れる自由変数の型は Element of [DOMAIN] という型まで拡張されます。 [DOMAIN] という記銘はその型が DOMAIN 型まで拡張される任意のオブジェクトを意味します。
Frænkel operator の場合には、変数の型は項の後にその型を書くことにより与えます。　その場合 *Frænkel operator* は：

{ *term* **where** *identifiers-of-variables* **is** *type* : *formula* }

という形になります。　*Frænkel operator* において一つ以上の型が現れた場合には **where** と : の間の表現はコロンで分離された回数に応じて繰り返されま

す。　式の中で与えられる型はその式の中でだけ参照されます。　例えば *reservation-of-variables* の構文の中では識別子 x は Real 型として予約されています。　一方、*Frænkel operator* 内ではその型は Nat に変更されます。そしてアーティクルの先のほうで型の指定がされていない場所での識別子 x を含む式の中では、x には変数の型の予約で割り当てられた型 Real が割り付けられます。

Frænkel operator を説明する例として、アブストラクト TREES-1 から Theorem 64 を使いましょう：

```
p ∈ T implies 「T, p, T1」= { t1 where t1 is Element of
        T : not p is_a_proper_prefix_of t1 }
              ∪ { p^s where s is Element of T1: s = s }
```

（訳注：現行の theorem :: TREES_1:64 は
p in T implies T with-replacement (p,T1) = { t1 where t1 is Element of
T : not p is_a_proper_prefix_of t1 }
¥/ { p^s where s is Element of T1 : s = s }; となります。）

識別子 p は FinSequence (//??Finsequence->FinSequence)型を持ち、一方識別子 T と T1 は Tree 型を持ちます。　シンボル {}, [ ] は曖昧さを持つ同音異義語（homonymous）で、その意味はそれが使用された文脈で変化します。　例えばカッコ {,} は 上記の(4)のもとでは n 対子の集合を意味し、n ≤ 8です。　そして上記の(5)のもとでは同じカッコが *Frænkel operator* を意味するために使われます。　シンボル[,] は(4)のもとでは、順序対（３対子、４対子）を意味するのに使われ、プライベートの述語、例えば P[x] ではカッコとして使われます。

set という語もやはり曖昧さを持つ同音異義語です。　一方では Mizar の型なのですが（**Ⅱ.4** を見て下さい）、他方では **set** … = … と **set of** … , という構文で現れます。　それらは、まったく異なった役割を演じます。

(6)．*identifier-of-functor (list-of-terms)* という形の表現は項です。
この種の項は、とくにローカル functor の定義の中で見出されるでしょう。

ローカル functor の２つの定義があります。　その中で識別子 x,y,z は型 Element **of** RATIONAL を持ちます。　その項は：

MULT(**set, set**) と UZUP(**set**)

です。

▷ **func** MULT (**set, set**) =
{ x・y: x ∈ $1 & y ∈ $2 & 0 ≤ x & 0 ≤ y } ∪ {z: z < 0}
▷ **func** UZUP (**set**) = { -x: **not** x ∈ $1}

(7) . ローカル定義のパラメータのシンボル $1, $2, $3, $4, $5, $6, $7, $8 は項です。 ローカル定義パラメータはローカル定義でのみ使用されます。

(8) . **it**は項です。

it は functor を定義する項（definienses）の中でだけ使用されます。 そこで **it** は functor の値を表します。 モードを定義する項（definienses）のなかでは、その例として与えられたモードの要素を表します。

functor を定義する項は、その functor の定義の中で、**means** という語の後に来る表現です。 それは、必要な変更を加えれば（*mutatis mutandis*：ラテン語）モードの定義の場合と同じになります。

項 **it** は、例えば下に示すようにシンボル Int を持つ functor の定義中に出現します。

definition
let P, 𝔊;
func Int P -> Subset **of** 𝔊 **means it** = (Cl (P^c))^c;
end;

(9) . **the** *symbol-of-selector* **of** *term* という型の表現は項です。

このようなフォームの項は*セレクター項（selector term）*と呼ばれます。

of につづく項の型は：

the *symbol-of-selector* **of** *term*

という表現で使われたセレクターのシンボルが現れる定義の構造体へかならず拡張されます（must expand）。

PRE_TOPC というアーティクルで導入される位相空間の構造体のなかに：

carrier
topology

というセレクターのシンボルが現れます.

項の例：

the topology **of** 𝔊、 **the** carrier **of** 𝔊

があります。

(10) . **the** *symbol-of-selector* という形の表現は項です。
この種類の項は、構造体のパターンの中でだけ、そしてセレクターのシンボルがちょうどそのパターンで以前にすでに導入されている時だけ現れます。
例として以下に、位相空間 TopStruct の構造体のパターンとして出現する項

the carrier

を挙げておきましょう。
≪ carrier->DOMAIN, topology->Subset_Family **of the** carrier ≫

(11) . *symbol-of-structure* ≪*list-of-terms*≫という形の記銘は項です。
この形の項は構造体の集合体（*aggregates of structures*）と呼ばれます。 シンボル≪、≫は標準ASCII集合にないので、シンボル(#, #)が代用されます。≪と≫の代わりに、それぞれ (# と #) が使用されますが、≪と#)あるいは(#と≫の組み合わせは使われません。
例として以下の：

TopStruct ≪REAL, RealTop≫

ような明確な構造を（でも構造体のパターンではありません）を与えることができます。 上記の構造体の場合、台集合(carrier)は REAL です。 定数 REAL は型 DOMAIN に拡張され、それは構造体 TopStruct の定義という観点から必要です。 RealTop（位相空間）は型 Subset_Family **of** REAL をもち、そのことは構造体 TopStruct の定義からの要求です。

(12) . *term* **qua** *type* という形の記銘は項です。
それは *qualified term（修飾された項）* とよばれます（//??a called->called 訳注：修飾子 qualifier とは区別が必要）。
例えば、P **qua** Subset **of the** carrier **of** 𝔊.
予約部の識別子 P は型 Subset **of** 𝔊 と予約されています。 しかし上の項ではその型はすでに、Subset **of the** carrier **of** 𝔊に拡張されています。

注釈： 語 ***qua*** *は型の拡張のみを行い、縮小はできません。*

(13) . カッコ (と) の中にあるもの、これもまた項です。
ここにカッコの中の項のいくつかの例を挙げます：
Cl (P ∪ Q)、Fr (P ∪ Q)、(P ¥ Q)^c、(k + 1)、(n + 1)
他の例は、式の議論をするときに示します。

Ⅲ.２．型（Types）

変数の識別子は必ず型を割り当てられています。　Mizar article では識別子の型が未知の変数の使用は許されません。　識別子の型はそれが出現する場所でローカルに設定（fixed）されるか、あるいはグローバルに予約で設定（fixed）されます（以下を見て下さい）。　ある種のモードは Mizar 言語では隠されて（hidden）います。　それ以外のもの（remaining ones）は特定（identified）されねばなりません。　型を構築するために、適切なモードを定義しないといけません。　モードのシンボルはかならずボキャブラリに含まれていないといけません。　以下にモードのシンボルを使う型をボキャブラリ HIDDEN（これは全ての Mizar article に自動的に付加されます）から提示します。

ここでは、それらは：

```
Any, set, Element of X, DOMAIN,
Subset of D, SUBDOMAIN of D,
Real, Nat.
```
（ここで X は型 set を持ちます）

となります。

これらの型の使用には Mizar article の作者が、ボキャブラリあるいは signature を環境部にインクルージョンする必要はありません、必要な signature とボキャブラリは自動的に結合付加されます。

注釈：型を書くときの重要な点は、シンボルを正確に、それらがボキャブラリのなかに見つかるはずの形で書くということです。　例えば識別子 K *を* DOMAIN *型に予約しようとしたときに、予約は：*

reserve K **for** DOMAIN;

という形になるべきで、例えば

reserve K **for** domain;

ではありません。

もう２つ例を：

Element **of** REAL と書くべきで、Element **of** real ではありません。

同様に Element **of** NAT と書くべきで、Element **of** nat ではありません。

型 Any は Mizar で最も広い型です、他のどの型もそれ（訳注：Any）にまで拡張されます。　その拡張された型 set は型 Any と等価（equal）です。　型 DOMAIN はいわゆる domains と呼ばれる non-empty sets を範囲に収めて

（range）おり； SUBDOMAIN of D はサブドメイン、すなわち non-empty subset を； Real は実数を； Nat は自然数を範囲に収めて（range）います。

さて、一般的な言葉では Mizar の型とは何であると言えるでしょう。

(1)． *Symbol-of-mode* **of** *list-of-terms* という形の表現は型です。

例：

Subset **of** G, Subset **of the** carrier **of** G, Point **of** G,
Subset-Family **of** G, Relation **of** X, Relation **of** X,Y

項のリストがゼロ・リストならモードのシンボルは型だけになるでしょう。 例えば、

Any, Real, TopSpace, FinSequence, Ordinal, Set-Family,
Relation, Function

などです。

(2)． 構造体のシンボル、例えば：

TopStruct, LattStr, IncStruct

は型です。

(3)． カッコの中の型もまた型です。

次の例で、今問題にしている型（type in question）はカッコの中にあります。

reserve P **for** (Subset of G), P **for** (Point of G), x **for** Any;
reserve S **for** (Subset_Family **of** G), r **for** Real;
reserve S **for** (Subset_Family **of** G), p **for** Subset **of** G;
let S **be** (SubSpace **of** G), P,Q **be** (Subset **of** G), x,y **be** Any;

上の例でカッコを取り除いて（omission）もエラーになりません。 しかし例えば：

reserve R **for** Relation **of** X, x **for** Any;

という記銘は正しくありません。 エラーは Relation **of** X がカッコの中に入っていないという事で起きます。 誰かが、なぜ Relation **of** X がカッコの中にないといけないかという質問を提出（might pose）するかも知れませんが。

さて、モード

Relation **of** *list-of-terms*

はゼロの項（Relation)、１つの項 Relation **of** X, そして２つの項を含むリスト Relation **of** X,Y に対して定義されています。　ここにそれに対応する定義があります。

definition
mode Relation -> **set means** x ∈ **it implies ex** y,z **st** x = [y,z];
end;

*(*x,y,z *は型* Any *を持ちます)*

definition
let X,Y;
mode Relation **of** X,Y -> Relation **means it** ⊆[:X,Y:];
end;

*(*X,Y *は* set *型です)*

definition
let X;
mode Relation **of** X **is** Relation **of** X,X;
end;

記銘 *Relation* **of** *list-of-terms* がカッコ（の中？）でないなら、*list-of-terms* は項の最大数である２からなります。　ですから記銘：

reserve R **for** Relation **of** X,x **for** Any;

は記銘：

reserve (R **for** Relation **of** X,x) **for** Any;

と同じと解釈されます。　しかし後の表現は不適当です。　その構文は正しくありません：　最後の **for** には項のリストが先行していません。

他方、モード；

Subset **of** *list-of-terms*
SubSpace **of** *list-of-terms*
Subset_Family **of** *list-of-terms*

は１つのそして１つだけの項を含むリストのためだけ対して定義され、それは定義で示されています。

```
definition
  let G;
  mode Subset of G is set of Point of G;
end;
definition
  let G;
  mode SubSpace of G -> TopSpace means
   Ω(it) c= Ω(G) &
   for P being Subset of it holds P∈the topology of it iff
   ex Q being Subset of G st Q ∈ the topology of G & P = Q ∩ Ω(it);
 end;
 definition
   let G;
   mode Subset_Family of G is Subset_Family of the carrier of G;
 end;
```

こうして、これらのモードの場合にはカッコは余計なものとなります。

(4). set of *type* の形の表現は型です。
この種の型はしばしばモードの定義に使われます。 例えば、

```
definition
    let G;
    mode Subset of G is set of Point of G;
end;
```

Mizar の型は木型の構造を持っています。 X は set 型まで拡張されます。しかし DOMAIN や SUBDOMAIN of ⌈DOMAIN⌉ には拡張されません。(訳注：⌈DOMAIN⌉ は原文のまま)

D は DOMAIN まで拡張されます。
D_1 は DOMAIN まで拡張されます。
S は SUBDOMAIN of D まで拡張されます。

Real は Element of REAL (REAL は DOMAIN です) の短縮型として導入され、一方 Nat は Element of NAT (NAT は SUBDOMAIN です) の短縮型として導入されます。

Element of S 型は Element of D 型まで拡張されるので、 Nat 型は Real 型まで拡張されます。

Ⅲ．３． 変数の予約（Reservation of variables）

以前 *Mizar article* では、型が与えられていない識別子の使用は許されないと述べました。 例文で示すように、識別子の型はそれが出現する場所で与えることができます。

- **for** A **being** Subset **of** 𝔊 **holds** A ⊆ Cl A;
- **let** x **be** Any, A **be** set;
- **reconsider** x **as** Real;

語 **being** と **be** は２つのうちのどちらかが使用されます、つまり Mizar という観点からはどちらでも無関係です。 しかし英語の文法にあわせるため、それらの使用に関するある慣例（規約）が採用されています。 Mizar 構文の中では **let ... be** が使用され、一方 **being** は量化式で使われます。

識別子を持つ変数が導入されるときには、いつでも識別子の型を指定するのを避けるため、変数の予約という Mizar 構文で変数の型をグローバルに設定することができます。 変数の予約は以下の形を持ちます。

reserve *list-of-identifiers* **for** *type*;

下に示す例は、今述べた構文の適用例です：

ⅰ．識別子 𝔊 が位相空間をその範囲に収める変数のつもりなら、識別子 𝔊 は TopSpace 型として予約されているはずです。 そのような予約は：

reserve 𝔊 **for** TopSpace;

という形を持ちます。

予約されてなくて、識別子 𝔊 が指定された意味で用いられるなら、識別子の型はその識別子を持つ変数が導入された時に、常にその場で与えらなければなりません。

ⅱ．識別子 P,Q が位相空間 𝔊 のサブセットをその範囲に収めるなら予約は以下のようであるはずです：

reserve P,Q **for** Subset **of** 𝔊;

いろいろな型を持つ変数の識別子を予約するなら、変数の予約の中の表現：

list-of-identifiers **for** *type*

を対応する回数だけ繰り返します。 該当する表現はコンマでお互いに区切られていなければなりません。

これは次の予約で説明しましょう：

reserve P,Q **for** Subset **of** 𝔊, x **for** Any, p **for** Point **of** 𝔊;

識別子 P,Q,x,p の対応する型への予約も次のようになります：

reserve P,Q **for** Subset **of** 𝔊
reserve x **for** Any;
reserve p **for** Point **of** 𝔊;

つまりいろいろな量化子に対し*変数予約構文*を分離して適用するということです。　しかし、それはテキストの不必要な増加を防ぐという観点からはベストの解決策ではありません。

例という形で与えられた他の予約は：

- 集合を表す変数の識別子 X の予約：
  reserve X **for** set;
- 実数を表す変数の識別子 X の予約：
  reserve X **for** Real;
- 部分集合 X を表す変数の識別子 Z と Y の予約：　ここで識別子 X は以前に集合型として予約、設定されています：
  reserve Z,Y **for** Subset **of** X;
- 実数集合の要素の識別子 x の予約：
  reserve x **for** Element **of** Real;

などです。

これらの構文以外の例は定理の証明のための章で示すことにしましょう。識別子を特定の型へ予約しておくことは、その識別子の最初の出現の前に行われているべきだという事は心に留めておかなければなりません。　予約というのは後になってするものでしょうか、すなわち識別子が数回現れた後では、その場でその識別子を持つ変数が導入されているので、もう識別子の型が導入されているはずです。

変数の予約はテキスト本文で行わないといけません。　しばしば予約はテキスト本文の頭部で、語 **begin** の直後で行われます。

同一の識別子に対して、構文 *reservation of variables* がアーティクルの著者の必要に従って数回適用されることもあります。　例えば最初に予約：

reserve x **for** Any;

を行い Mizar article の後の部分で識別子 x の型を予約：

reserve x **for** Real;

に従って Real に変えます。　この２つの予約の中間で識別子 x の型が与えられていないところでは、出現する変数 x はすべて最初の予約で割り当てられた型、つまり Any を持ちます。　そして２番目の予約以降では、型が与えられていない場所で出現する識別子 x はすべて型 Real を持ちます。

さて Mizar の式の話に進みましょう。

Mizar では次のような式の類別（classification）があります。

- 原子式（atomic formulas）、
- 統語接続子（sentential connectives）により原子式から形成された式、
- 量化式（quantified formulas）

Ⅲ.４．原子式（atomic formulas）

数種類の原子式があります。

(1)．　述語式（A PREDICATIVE FORMULA）　すなわち：

list-of-terms　symbol-of-predicate　list-of-terms

の形をした表現は原子式です。　ここに位相空間に関するアーティクルで使われた述語のシンボルをいくつか挙げます：

is_open, is_closed, is_open_closed, is_dense,
is_boundary, is_nowheredense, ⊆, are_separated.

引数の数を（右側のも左側のも）、述語のそれぞれの場合について次のテーブルに示します。

述語のシンボル	原子式	左側引数の数	右側引数の数
is_open	P is_open	1	0
is_closed	Q is_closed	1	0
is_open_closed	Q is_open_closed	1	0
is_dense	A is_dense	1	0
is_boundary	B is_boundary	1	0
is_nowheredense	P is_nowheredense	1	0
⊆	P ⊆ Q	1	1
are_separated	A,B are_separated	2	0
⊨	D,f ⊨ H	2	1

識別子 D は集合の族(a family of sets)、f はその族の要素による評価(valuation)を意味し、H は ZF 集合論の言葉で任意の式を意味します。 テーブルの最後の式は集合の族 D において式 H は f の評価（valuation）を満たすことを示しています。 原子式の例は上のテーブルにあります。 それ以外の例を挙げます。

```
Cl P ¥ Cl Q ⊆ Cl (P ¥ Q)
Int P is_open
skl p is_connected
B is_a_component_of 𝔊
𝔉 is_a_cover_of 𝔊
P,q are_joined
p ∈ skl p
```

特に、記号 = は述語のシンボルなので：

term = *term*

という形の表現は*等価式（equality formula）*と呼ばれる原子式です。 *等価式*の例は：

```
Fr(Fr(Fr P )) = Fr(Fr P)
Cl(Int P) = Cl(Int(Cl(Int P)))
P^c = Ω𝔊 ¥ P
P^c = (P qua Subset of the carrier of 𝔊)^c
P ¥ Q = P ∩ Q^c
Cl (Cl P) = Cl P
```

などがあります。 同様に：

term <> *term*

の形の表現は原子式で、記号 <> は述語のシンボルです。

例： P ∩ Cl Q <> Ø, Int (Cl Q) ∩ Int (Cl P) <> Ø.

(2) . *identifier-of-predicate* [*list-of-terms*] という形の表現は原子式です。この種の項はプライベート述語の場合にだけ出現します。 その述語の識別子はどのボキャブラリにもリストアップされていません。

(3) . *term* **is** *type* (//??typ->type) という形の表現は原子式です。
その形の式は*修飾する式*（*qualifying formula*）と呼ばれます。

修飾する式の例：

```
x is Point of G
x is set
```

定理 NAT_1:1 を挙げます：

```
x is Nat implies x+1 is Nat
```

ここで識別子 x は型 Real を持ちます。
上記の含意の前項も後項も今考えている種類の原子式です。

Ⅲ．5　命題結合により原子式から形成された式 （Formulas formed of atomic formulas by propositional connectives）

Mizarにおいては下記のシンボル：

not,　&,　or,　implies,　iff,　contradiction

は統語接続子（sentential connectives）を表すのに使用され、それぞれ、否定、論理積（合接、共通集合)、論理和（離接、合併集合)、含意、同値、矛盾を意味します。
contradiction は引数ゼロの統語接続子です。

*注釈：***contradiction** *はMizarでは式* **thesis** *と同じように扱われます。*

さて **not** が一番強い結合力を持っています、結合力は &、 次に **or**、 次に **implies** と **iff** が同じ程度の強さを持ちます。　しかし **implies** と **iff** の結合力は量化子の結合より強力です。

Mizar では **implies** と **iff** の結合力が同じなので、その２つが同時に式の中に現れる時には接続子 **implies** と **iff** の引数を示すためにカッコ （ と ） の使用が必要になります。

Φ_1、Φ_2、Φ_3を原子式としましょう。　式

$$\Phi_1 \text{ implies } \Phi_2 \text{ iff } \Phi_3$$

は接続子 **implies** と **iff** がどの引数を持つか不明なので、不適格（ill-formed）な記述です。　さらに、Mizarにおいてはカッコ （ と ） は算術でのそれと類似のはたらきをします。　つまりそれらは演算の実行の順序を指示します。

統語接続子（sentential connectives）により形成された式の例は：

```
P is_closed & Q is_closed implies Cl(P ∩ Q) = Cl P ∩ Cl Q,
```

P is_open **iff** Fr P = Cl P ¥ P,
P ∈ P^c **iff not** p ∈ P,
P ∩ Ø𝒢 = Ø & Ø𝒢 ∩ P = Ø,
(A is_closed & B is_closed) **or** (A is_open & B is_open)
implies A ¥ B, B ¥ A are_separated,
x ∈ P **implies** x **is** Point **of** 𝒢
A is_connected & A ⊆ B ∪ C & B,C are_separated
implies A ⊆ B **or** A ⊆ C

などがあります。

Ⅲ．６． 量化式（Quantified formulas）

量化式の議論に進む前に、修飾された変数のことをしらべてみましょう。

Mizar articleではしばしば修飾された変数のリスト（*list of qualified variables*）と呼ばれる記銘があります。　そのリストの例は：

x,
A, K, n,
P, Q **being** Subset **of** 𝒢,
𝒢 **being** TopSpace, 𝒮 **being** SubSpace **of** 𝒢,
ℱ **being** Subset-Family **of** 𝒢, p **being** Point **of** 𝒢, x, y.

などがあります。　一般的には、修飾された変数のリストは下記に示した３つの形の表現のうちの一つから構成されます：

●●● *暗黙的に修飾された変数（variables-qualified-implicitly）*
この名前は変数の識別子のリストのことです、すなわち変数の識別子の有限の列で、お互いがコンマで区切られています。　例：

C　　x, y, p　　A, B

見ればわかるようにその場合には識別子の型は同定（not identicated）できません。　これは変数の予約のための識別子のリストから採られたことを意味します。

●●● *明示的に修飾された変数（variables-qualified-explicitly）*
最初に

segment-of-qualified-variables（修飾された変数のセグメント）

を説明しましょう。　それは *list-of-identifiers-of-variable* **being** (or **be**)

qualification という形の記銘です。

*qualification（修飾）*は修飾された変数（qualified variables）のセグメントに現れる変数の識別子の型です。

もちろん、その表現で指示される型は、その中に出現するの全ての量化子（quantifiers）と同じです。

修飾された変数のセグメントの例：

```
a being Any
a,b,c,d be Any
m,n be Point of 𝔊
P,Q being Subset of 𝔊
```

明示的に修飾された変数（*variables-qualified-explicitly*）はコンマで分離された修飾された変数のセグメントの有限の列です。　もっとも単純なケースは単一のセグメントです。

明示的に修飾された変数の例：

```
x1,y1 be Any
```
（これは修飾された変数の単一のセグメントです）

```
x being Real, X being set
```
（この場合修飾された変数の２つのセグメントがあります）

```
𝔊 being TopSpace, 𝔖 being SubSpace of 𝔊, p being Point of 𝔊
```
（この場合修飾された変数の３つのセグメントがあります）

●●● *明示的、暗黙的に修飾された変数*（*variables-qualified-explicitly*、*variables-qualified-implicitly*）

例：

```
P, Q being (Subset of 𝔊),   x, p
```

variables qualified explicitly (one segment of qualified variables)

variables qualified implicitly

```
P being (Subset of 𝔊), p being (Point of 𝔊),   x, y, z
```

variables qualified explicitly (two segments of qualified variables)

variables qualified implicitly

明らかに、型が与えられていない上の例の変数の識別子は変数の予約に現れた量化子のリストから採られたのに違いありません。

注釈： 順番の入れ替え（暗黙的に修飾された変数を明示的に修飾されたものに先行させる）は不可能です。 なぜなら全ての変数は明示的に修飾された変数になるかも知れないからです。

上記の警告を説明するため、例という方法で、以下の式を考えて見ましょう：

（●）**for** A **being**(Subset **of** 𝔊), x **st** x ∈ A **holds** x is_Point **of** 𝔊

この式で変数 A は明示的に修飾され、一方変数 x は暗黙的に修飾されています。 x の型は予約部で設定されているのは間違いありません。 この場合は型 Anyとなります。 今、この式の変数 A と x の出現の順番を変えると、つまり、はじめに暗黙的に修飾された変数を与え、次に明示的に変数を修飾した式を与えると：

for x, A **being** Subset **of** 𝔊 **st** x ∈ A **holds** x is_Point **of** 𝔊

を得ます。
しかしこの式は変数 x と 変数 A を明示的に修飾した変数になってしまっています； この過程で識別子 x の型は Subset **of** 𝔊 に変わってしまっています。 しかし、それは本来 Anyであるべきです。
あるケースでは変数の交換が実行されます。 これは上の（●）が記された式の例で説明しましょう。 その場合

for A **being** (Subset **of** 𝔊), x **st** x ∈ A **holds** x is_Point **of** 𝔊

はMizarでは同じ意味を持つ式に置き換えられるかもしれません。 ここにその式：

for A **being** Subset **of** 𝔊 **for** x **st** x ∈ A **holds** x is_Point **of** 𝔊

を示します。
さて、これで量化式（quantified formula）に行くことができます。
量化式（これは全称文（universal sentence）とも呼ばれます）は量化子がオープンに（occurs openly）現れ、主文を形成する functor の式です。
Mizarにおいては次のシンボル使用法が量化子のために採用されています：

for ... holds ... - 全称量化子（universal quantifier）のため
for ... st ... holds - 純化全称量化子（purified universal quantifier）のため
（すなわち、修飾されたスコープを持つ全称量化子）

ex … st …　　- 存在量化子のため。

修飾された変数（qualified variables）のリストはいろいろな形をとりますが、全称文、それは全称量化子が現れる文のことですが、は次の形となるでしょう。

(A)．　**for** *identifiers-of -variables* **holds** *formula*

量化式の例：

for P **holds** P ⊆ Cl P

（位相空間 G *の任意の部分集合* P *について* P ⊆ Cl P *が成り立つ，*
あるいは（さもなければ）*位相空間* G *の任意の部分集合はその閉包に含まれる）*

for P,Q **holds** Cl (P ∪ Q) = Cl P ∪ Cl Q

（位相空間 G *の任意の２つの部分集合* P,Q *について* Cl (P ∪ Q) = Cl P ∪ Cl Q *が成り立つ）*。

上の各ケースで変数の識別子の型がオープンに与えられていないのがわかります。　Mizar articleでそのような式を使う時は、事前にそれら変数の識別子の予約を変数の予約として行うべきであること心にとめておくべきです。

しかし早い時期に量化式のなかに現れる変数の予約はあとでとりやめる（abstain）かも知れません。　その場合には変数の識別子の型は式が書かれた時点で設定されねばなりません。　量化式の形は次のようになるでしょう：

(B).　**for** *segment-of-qualified-variables* **holds** *formula*

量化式はこのような形を持ち、なかでも予約されていない変数の識別子は単一の同じ型を持ちます。　さもなければ **for** と **holds** の間の表現はその回数に応じて繰り返され、コンマで区切らねばなりません。

もう一つの場合を考えて見ましょう。　今、量化子に現れる変数の識別子が以前にある型（corresponding types）で予約されたことがあるなら、式を書いている最中でなければ、この同じ変数の修飾法（qualifiers of variables）を他の型に適用したいと考えます。　その式は上記の (B) に続く表現になるでしょう。

例で説明しましょう。　以下の予約が与えられているとしましょう：

reserve A **being** SubSpace **of** G

この式で位相空間 G の部分集合である識別子 A を使いたいと思います。　すると識別子 A の新しい型 (//??o->of) は次の式で与えられます：

for A **being** Subset **of** 𝔊 **holds** A is_closed **iff** Cl A = A

量化式（quantified formula）の構造（structure）を説明する例：

1）**for** P **being** Subset **of** 𝔊 **holds** P ⊆ Cl P
（*位相空間* 𝔊 *の任意の部分空間* P *について* P ⊆ Cl P *が成立*）

2）**for** P,Q **being** Subset **of** 𝔊 **holds** Cl (P ∪ Q) = Cl P ∪ Cl Q
（*位相空間* 𝔊 *の任意の部分空間* P, Q *について* Cl (P∪Q) = Cl P ∪ Cl Q *が成立*）

3）**for** A **being** (Subset of 𝔊), x **being** Any **st** x∈A **holds** x is_Point **of** 𝔊
（*位相空間* 𝔊*の任意の部分集合* A、*その部分集合* A *の要素である任意のオブジェクト* x *について、*x *は位相空間* 𝔊 *の点である，が成立*）

例の３）の定理はまた、このように記述されます：

for A **being** (Subset of 𝔊) **for** x **being** Any **st** x∈A **holds** x is_Point **of** 𝔊

例（EXAMPLES）：

- **for** A **being** (Subset **of** 𝔊), p **being** Points **of** 𝔊 **holds** p∈Cl A **iff** **for** C being Subset of 𝔊 **st** C is_closed **holds** (A⊆C **implies** p∈C)
- **for** 𝔖 **being** (SubSpace of 𝔊), P,Q **being** (Subset **of** 𝔊), P1, Q1 **being** Subset **of** 𝔖 **st** P = P1 & Q = Q1 & P ∪ Q ⊆ Ω∅ **holds** P,Q Are_separated **implies** P1, Q1 are_separated
- **for** 𝔖 **being** (SubSpace **of** 𝔊), P **being** (Subset **of** 𝔊), Q **being** Subset **of** 𝔖 **st** p ≠ ∅𝔊 & P = Q **holds** A is_connected **iff** B is_connected

さらに量化式は：

(C). **for** *variables-qualified-explicitly*， *variables-qualified-implicitly* **holds** *formula*

という形を持ちます。 純化量化子（purified quantifier）を持つ量化式もあります。 その式は：

for *list-of-qualified-variables* **st** *formula* **holds** *formula*

という形となります。 純化量化子（purified quantifiers、制限されたスコー

プを持つ量化子）を持つ式の構造は例で説明します。　それを提示する前に、位相空間に関係するアーティクルで導入されるモードと述語について述べ（specify）ましょう。

それらはモード：

SubSpace **of** 𝔊,
Subset-Family **of** 𝔊

そして述語：

P,Q are_separated,
𝔊 is_connected,
p, q are_joined

です。
上記のモードのフォーマットはそれぞれ位相空間 𝔊 の部分空間と位相空間の部分集合の族（family）を表すために導入されました。　上記の述語については以前、議論しました。 さあ例に行きましょう。

(i) **for** P,Q **st** P ⊆ Q **holds** Cl P ⊆ Cl Q
*(*P ⊆ Q*である位相空間* 𝔊 *の任意の部分集合* P,Q*について* Cl P ⊆ Cl Q *が成立する)*　　(//?? P⊆Q -> Cl P⊆Cl Q)

(ii) P is_boundary **iff** (**for** Q **st** Q ⊆ P & Q is_open **holds** Q = Ø)
(P*は境界点の集合であり、かつその場合に限り、その* P *に含まれる任意の開集合* Q *について、*Q = Ø *が成立する。)*

もし識別子 P と Q の予約がされていなかったとすると、上記の式は以下のようになります：

for P **being** Subset **of** 𝔊 **holds** P is_boundary **iff**
(**for** Q **being** Subset **of** 𝔊 **st** Q ⊆ P & Q is_open **holds** Q = Ø)

(iii) **for** 𝔖 **being** (SubSpace **of** 𝔊), P_1, Q_1 **being** (Subset **of** 𝔊), P, Q **being** Subset **of** 𝔖 **st** P = P_1 & Q = Q_1 **holds** P, Q are_separated **implies** P_1, Q_1 are_separated
(位相空間 𝔊 *の任意の部分集合* 𝔖 *そして位相空間* 𝔊 *の部分集合* P_1, Q_1 *そして部分空間* 𝔖 *の* P = P_1 *かつ* Q = Q_1 *であるような部分集合* P, Q*について：　*P, Q *が分離（separated）ならば* P_1, Q_1 *も分離である、が成立)*

(iv) **for** 𝔖 **being** (SubSpace **of** 𝔊), A **being** (Subset **of** 𝔊), B **being** Subset **of** 𝔖 **st** A ≠ Ø 𝔊 & A = B **holds** A is_connected **iff** B is_connected

（*位相空間* 𝔊 *の任意の部分集合* 𝔖 *そして* A≠∅𝔊 & A = B*であるような位相空間* 𝔊 *の部分集合* A *そして部分空間* 𝔖 *の部分集合* B *について：* A *は連結（connected）で、かつその時に限り* B *は連結である、が成立）*

存在量化子（existential quantifier）を含む式は下記リストの３つの形：

- **ex** *variables-qualified-implicitly* **st** *formula*
- **ex** *variables-qualified-explicitly* **st** f*ormula*
- **ex** *variables-qualified-implicitly*,
  variables-qualified-implicitly **st** *formula*

の１つを持ちます。

下の例は存在量化子（existential quantifier）を含みます。

```
x ∈ Int P iff ex Q st Q is_open & Q ⊆ P & x∈Q,
```

```
(ex x being Point of 𝔊 st for y being Point of 𝔊 holds x,y
are_joined) iff (for x,y being Point of 𝔊 holds x,y are_joined).
```

存在量化子を持つ式の他の例は後でテキストに出てきます。　ここでは４個の位相空間の定理をはじめに英語で、ついで Mizar記法で書いたものを示します。

１） P は境界集合（boundary set）でかつ、その場合に限り自分自身の境界にふくまれる。

２） P⊆Q であるような位相空間 𝔊 の任意の部分集合 P, Q について Cl P ⊆Cl Q が成立する。

３） 位相空間 𝔊 の任意の部分集合 A は Cl A = A の場合、その場合に限り閉集合である。

４） 点 p は集合 P の境界にある場合、その場合に限って、そこの p∈Q である任意の開集合 Q について： 集合 P と集合 Q の共通集合は空集合ではなく P の補集合と集合 Q の共通集合も空集合ではない、が成立する。

以下は上の定理を Mizar記法で書いたものです。

１） `P is_boundary` **iff** `P ⊆ Fr P`

２） **for** `P, Q` **being** `Subset` **of** 𝔊 **st** `P ⊆ Q` **holds** `Cl P ⊆ Cl Q`

3） **for** A **being** Subset **of** 𝔊 **holds** A is_closed **iff** Cl A = A

4） p ∈ Fr P **iff**
(**for** Q **st** Q is_open & p ∈ Q **holds** P∩Q≠Ø & P^c∩Q≠Ø)

これまでの例は、その大部分が全称（universal）か存在（existential）かにかかわらず量化子１つの式に関するものです。　しかし式によっては1つ以上の量化子が現れます。　以下の例で見ることができます。

for A **being** (Subset **of** 𝔊), p **being** Point **of** 𝔊 **holds** P ∈ Cl A **iff for** G **being** Subset **of** 𝔊 **st** G is_open **holds** p∈G **implies** A ∩ G ≠ Ø𝔊
(位相空間 𝔊 の点 p は位相空間 𝔊 の閉包 A にある場合、その場合に限り点 p を含む位相空間 𝔊 の任意の開集合 G について G と A の共通集合（intersection）は空集合ではない)、

P is_open **iff** (**for** x **holds** x∈P **iff ex** Q **st** Q is_open & Q⊆P & x∈Q)
（ P は開集合である場合、その場合に限り任意の x が x∈P である場合、その場合に限り Q⊆P かつ x∈Q である開集合 Q が存在する)、

P is_closed **implies** (P is_boundary **iff for** Q **st** Q ≠ Ø & Q is_open **ex** G **st** G⊆Q & G≠Ø & G is_open & P ∩ Q = Ø)
（ P が閉集合ならば、P は境界集合である場合、その場合に限り Q ≠ Ø かつ開集合である任意の集合 Q について、 G⊆Q かつ G≠Ø かつ G は開集合かつ P∩Q = Ø である集合 G が存在する)、

for 𝔖 **being** Subset-Family **of** 𝔊 **st** 𝔖≠Ø & **for** A **being** Subset **of** 𝔊 **st** A∈𝔖 **holds** A is_closed **holds** meet 𝔖 is_closed
(位相空間 𝔊 の空でない任意の部分閉集合の族 𝔖 の共通部分は閉集合である)

for 𝔖 **being** Subset-Family **of** 𝔊 **st** (**for** A **being** Subset **of** 𝔊 **st** A∈𝔖 **holds** A is_connected) & (**ex** A **being** Subset **of** 𝔊 **st** A≠Ø(𝔊) & A∈𝔖 & (**for** B **being** Subset **of** 𝔊 **st** B∈𝔖 & B≠A **holds not** A,B are_separated)) **holds** union 𝔖 is_connected
（ 𝔖 を位相空間 𝔊 の連結部分集合の任意の集合族として、そのうちの一つは空集合ではなく、その集合族のいかなる要素とも分離ではない。　そのときその集合族の要素の合併集合（union）は連結集合である）。

meet と union は各々が１つの引数を持つ functorのシンボルで（最後の式の記述は右側引数がその唯一の引数です）、それぞれ位相空間の部分集合族の共通集合と合併集合を意味します。

これまでの例はアーティクル PRE_TOPC, TOPS_1 そして CONNSP_1 から採録され、位相空間に関する問題に関係することはおのずから明らかです。

もう一度純化量化子(purified quantifier)を持つ一般化文(general sentence)に戻りましょう。 もちろん、その文は意味を変えることなく別の流儀で記述することができます。 一般化文の量化子の範囲の制限の撤廃と含意（implication）による条件の置き換えのことを言っています。

Φ_1、Φ_2を任意の式としましょう。 一般化文（その中に全称量化子を持つ文です）:

for *list-of-qualified-variables* **st** Φ_1 **holds** Φ_2

は文

for *list-of-qualified-variables* **holds** Φ_1 **implies** Φ_2

と等価（equivalent）です。

さて式

Φ_1 **implies** Φ_2

にはカッコがありません。 **implies**（**iff**も同様ですが）の結合は量化子のそれよりも強いからです。

例：

式

for P,Q **st** P⊆Q **holds** Cl P ⊆ Cl Q

は Mizar にとって式

for P,Q **holds** P⊆Q **implies** Cl P ⊆ Cl Q

と同じ意味を持ちます。 この２つの式は同一の semantic correlate を持つからです（**Ⅲ.7.**を見て下さい)。

同様に式：

P is_boundary **iff** (**for** Q **st** Q⊆P & Q is_open **holds** Q = ∅)

と

P is_boundary **iff** (**for** Q **holds** Q⊆P & Q is_open **implies** Q = ∅)

はMizarにとって同じ意味を持ちます。

全称量化子をオープンに（openly）持つ定理は量化子を持たない式（non-quantified）に書き下すことができます。

例えば、定理：

for P **being** Subset **of** 𝔊 **holds** Int P = P ¥ Fr P

は：

Int P = P ¥ Fr P

と書き下すことができます。 両方の式は Mizar にとって同じ意味を持つからです（semantic correlatesを見て下さい）。

同様に文：

for 𝔊 , P **holds** Int P = (Cl (P^c))^c,
for P **holds** Int P = (Cl (P^c))^c,
Int P = (Cl (P^c))^c,

はシステムにとって全て同じものとして読み込まれるでしょう（𝔊 , P が前もって設定されていない場合です）。

式：

Int P = (Cl (P^c))^c

は

for 𝔊 , P **holds** Int P = (Cl (P^c))^c

とシステムに読み込まれるでしょう。

多様な式の書き方はそのアーティクルの作者にとってのみ意味があります。それらのうちのいくつかは、より読み易いでしょう、しかしPC Mizarの言語プロセッサは式を読んである決まった形(//??for->form)に変換します（Ⅲ. 7 を見て下さい）。

ここで Mizar 式の異なった形の書き方を説明するため他の例を挙げます：

1） **for** P,Q **st** P ⊆ Q **holds** Cl P ⊆ Cl Q

は：

P ⊆ Q **implies** Cl P ⊆ Cl Q と書くことができます。

2） **for** P,Q **st** P is_dense & Q is_dense & Q is_open **holds** P ∩ Q is_dense

（ P, Q が事前に指定されていない場合）

は：

P is_dense & Q is_dense & Q is_open **implies** P ∩ Q is_dense

と書くことができます。 含意命題の前項にはカッコがありません、なぜなら論理積（conjunction）は含意よりも強い結合力を持つからです。

3） **for** P **st** P is_open **holds** Cl(Int(Cl P)) = Cl P

は：

```
P is_open implies Cl(Int(Cl P)) = Cl P
```

と書くことができます。

新しいバージョンで書かれた全ての例で、量化子は（is understood 訳注：システムにより）理解されます。 上の例の式は、量化子を使って書かれてなければ、システムによって量化式に加工されて処理されるでしょう（Ⅲ.7 を見て下さい）。 変数の識別子は **for** の後に続きます、式 `P ⊆ Q` **implies** `Cl P ⊆ Cl Q` の中では `P` の識別子が最初にきて、`Q` の識別子が（空間 `G` が前もって設定されていなければ）これに続きます。 さらに `G` の識別子は量化式に処理され、その量化式の中では語 **for** が最初で次に `P` の識別子が続き（あるいは識別子 `G` と `P`）、そしてつぎに上記 (1) の `Q` の識別子が続きます。 しかし要求される識別子の順序が自動的に配置された順序とは異なることがあります。 そのようなケースでは式は望ましいように量を定められた（desired quantified）量化式のバージョンで書かれなければなりません。（//??quantified の重複）。

語 **ex** あるいは語 **for** の前の語 **hold** は省略（omit）することができます。

式：

for..... **holds** **ex**.....

は下に示すように **hold ex** という表現を語 **ex** で置き換えて：

for..... **ex**.....

と記述しても良いのです。

例：

式：

1) **for** `A` **being** `Subset` **of** `G` **st** `A ≠ ∅ G` **holds ex** `x` **being** `Point` **of** `G` **st** `x∈A`

2) `P is_closed` **implies** (`P is_boundary` **iff for** `Q` **st** `Q≠∅ & Q is_open` **holds ex** `G` **st** `G⊆Q & G≠∅ & G is_open & P ∩ G = ∅`)

は、今までのべたことに従って、次のような様式：

1) **for** `A` **being** `Subset` **of** `G` **st** `A ≠ ∅ G` **ex** `x` **being** `Point` **of** `G` **st** `x∈A`

2) `P is_closed` **implies** (`P is_boundary` **iff for** `Q` **st** `Q≠∅ & Q is_open` **ex** `G` **st** `G⊆Q & G≠∅ & G is_open & P ∩ G = ∅`)

にも書かれるでしょう。 同様に：

for.....**holds for**.....

という形の式は：

for..... for.....

と記述することもできます。 ここで **hold for** という表現は語 **for** で置き換えられています。

例えば：

for 𝔖 **being** SubSpace **of** 𝔊 **holds for** A **being** Subset of 𝔖
holds A **is** Subset **of** 𝔊 は：

for 𝔖 **being** SubSpace **of** 𝔊 **for** A **being** Subset of 𝔖
holds A **is** Subset **of** 𝔊

で置き換えても良いのです。

ここまでの例は、定理は複数の方法で記述できることを示しています。 記述の形はアーティクルの作者の選択によります。 定理の内容の記述法は、最も可読性が高く実用的であるものが推奨されます。 例えば純化量化子に全称文の使用（語 **st** の使用による（//??trough->through））はしばしば可読性を向上させます。 同様のことが式を書く時の変数の識別子の型指定にもあてはまります。 変数の予約はアーティクルのはじめの部分か、あるいは後方でテキストの中で行われます。 アーティクルが長い場合は（その中に含まれる）定理を読むとき、変数の型が指定されていないと、テキストの（訳注：初めの部分の）予約を見る必要があります。 それは無意味な時間のロスを意味します。

Ⅲ.７． Semantic correlates

PC Mizar の言語プロセッサは読み込む式（項、型）をある種の標準形に変形します。 式の変形で得られる標準型は式の semantic correlate（semantic form）と呼ばれます。 前もって式（項、型）の変形を可能にする同値関係（equivalence）が式について定義されています。 それは、（訳注：同値）関係を持つ２つの式は同じものに変形されると述べています。 この同値関係の抽象化クラスは semantic correlates と呼ばれます。 ２つの式が同一の抽象化クラスに入るなら、同じ semantic correlates を持ちます。 抽象化クラスを形成する式のなかからその抽象化クラスの標準的表現を選ぶことができます。そのような式は、否定（**not**）、論理積（**&**）、**not contradiction**、すなわち VERUM（訳注：ラテン語「真性の」）の記号、そして基本文（base sentences）、すなわち原子式と全称文で構成されます。

さらに論理積（conjunction）と否定（negation）は以下の条件を満たします。

１．　論理積は結合律を満たします（associative)、つまり任意の式 α_1、α_2、α_3 について

$$(\alpha_1 \ \& \ \alpha_2) \ \& \ \alpha_3 \quad と \quad \alpha_1 \ \& \ (\alpha_2 \ \& \ \alpha_3)$$

は同じ抽象化クラスに属します。　それらは同一の semantic correlates を持ちます。

２．　否定は退縮（involution）です。　α の任意の式について式　not not α と　α は同一の semantic correlate を持ちます。

３．　量化子によりオープンに（openly）結び付けられていない変数の、自由変数が式に出現したときには、全称量化子が自動的にその式の前に付けられます。

例えば式 α(x) を書く時、その中で x は自由変数です（すなわち量化子により結合されていません)、そしてその式はシステムによって式 **for** x **holds** α(x) の形に読み込まれるでしょう。　こうして式

$$\alpha(x) \qquad と \qquad \textbf{for}\ x\ \textbf{holds}\ \alpha(x)$$

は同一の semantic correlate を持つことになります。

式 **contradiction** はその semantic form として **not** VERUM を持ちます。

term = *term*

の形の式以外の述語式（predicative formulas）の semantic correlates は同一の原形（初期型）です（原文：original(initial) form)。

term <> *term*

の形の述語式（predicative formula）の semantic correlates は式

not *term* = *term*

です。

term <> term

の形の式は

term = term

の形の式の反意語（antonym）です。

さらに、式 x≤y（ここで x,y は型 Element **of** REAL を持ちます）については、:

x > y　　と　　y < x

の２つの反意語があり、この２つは同意語（synonym）です。　y≥x は（訳注：x≤y の）同意語です。

semantic correlates の知識は証明のスケルトンの構築に使えます、式の semantic correlates の形はその式の証明のスケルトンを決めるからです。

もし P,Q,A,B,C, G が定数でなく、早い時期に設定されている変数の識別子であれば、式

a)　P ⊆ Cl P　　と　　**for** P **holds**　P ⊆ Cl P

b) Int Q is_open　　と　　**for** Q **holds** Int Q is_open

c) Ω(G) ¥ A = B ∪ C & B,C are_separated & A is_closed **implies** A ∪ B is_closed & A ∪ C is_closed

と

for G, A,B,C **holds** Ω(G) ¥ A = B ∪ C & B,C are_separated & A is_closed **implies** A ∪ B is_closed & A ∪ C is_closed

はそれぞれ、 a), b), そして c) において、それぞれに対応する同一の semantic correlates を持ちます。　式の中で α と β が同一の semantic correlates を持てば、α は β で置き換えられるし、その逆もできます。　これには利点があります。　α を証明したいとき、しばしば β を証明するほうが便利なことがあるからです。

ここに式のペアを置きます：

α & **not contradiction**	と	α
α **implies contradiction**	と	**not** α
not contradiction implies α	と	α
for x **for** y **holds** α	と	**for** x, y **holds** α
ex x **st ex** y **st** α	と	**ex** x, y **st** α
for x **st** α **holds** β	と	**for** x **holds** α **implies** β
α & β **implies** γ	と	α **implies** (β **implies** γ)
not not α	と	α
α **or** β	と	**not** α **implies** β
not ex x **st** α	と	**for** x **holds not** α
α **iff** β	と	(α **implies** β) & (β **implies** α)

各ペアの式は Mizar にとって同じ意味を持ちます。　それらは同一の semantic correlates を持ちます。

注釈：文　α & β　と　β & α　は異なる semantic forms を持ちます。
*α **or** β　と　β **or** α　にも同様のことが言えます。*

上の例で式 α、β、γ は、正しい構文と、与えられた文の枠組み（schemata）に収まることを（subsumption）確保するために、必要な時はいつでもカッコのなかにいれるべきです。

例えば、γ は含意でしょうか、それとも同値でしょうか。 それが現れる式は：

α & β **implies** γ　　と　　α **implies** （β **implies** γ）

と書かれるべきなのです。
同じことが α と β にも適用されます。

Ⅳ. Mizarでの文の証明（PROVING SENTENCES IN MIZAR）

Ⅳ.1. ジャスティフィケーション（Justifications）

Mizar 記法における定理に進む前にそれの justify に進みましょう。定理の justifying にはいくつかの可能性があります。 しかしこの時点ではその一つ（//??one them->one of them）について述べましょう、 すなわち straightforward justification で、これは参照（justifyされる定理の前提（premises）を指示するラベルのリスト）を与える方法です。 straightforward justification は：

a）simple justification
b）schemaによるjustification

に分類することができます。

direct justification は次の形を持ちます：

(*) *sentence-justified* **by** *list-of-references*;

参照のリストはコンマでお互いに分離されたひと続き（sequence）の有限の参照です。

参照については既に議論しました。 それらはライブラリ参照（（訳注：他の）アーティクルで見つけられる定理）とローカル参照（アーティクル内で既に justify されている文でラベルを通じて見つけて使用することができる定理）に分類されることに注意してください。

以下に直接法で justify されたいくつかの文を載せます：

（1） `M ∪ Ø = M` **by** `BOOLE:60;`
`BOOLE:60`はライブラリ参照です。 それはファイル`BOOLE.abs`のなかにある定理`No.60`を意味します。
（2） `K + 1 = 1 + k` **by** `NAT_1:3;`
（3） `K ≤ 0 & 0 ≤ 1` **implies** `K ≤ 1` **by** `NAT_1:13;`
（ファイル`art.lst`の例`No.2`を見て下さい。 ：欠落）。

注釈： **by** *による justification は以前に現れた文（テキストの前の部分あるいは前のアーティクル）のラベルを含んでいないといけません、そして参照の場所（つまり、前にすでに閉じた推論（1）のレベル、あるいは現在の推論（2）のレベル）からアクセスできないといけません。 そこで次の例の justification は正しくないと考えられます（would be incorrect）。*

例

```
for M,N being set, x being Any st x∈M holds x ∈ M ∪ N
proof
    let M,N be set, x be Any;
    assume A: x∈M;
    hence thesis by BOOLE:8;
end;

for x being Any, M being set holds x∈M by A;
                                                    *(1)
B: now
    let x be Any, M be set;
    x∈M implies x = x by B;
end;                                    * (2)
```

（art.lstの例No.3を見て下さい。：欠落）

ある種の定理の場合、証明に進む前に１つあるいは複数の補助的な補題（//??a->an）を証明しておくことは、なかなか便利な方法です。 それは straightforward justifications になるので、本来の定理の証明に、なんの問題も生じません。

例えば、同値（equivalence）の定理を証明したければ、事前に必要な含意の証明をしておくことができます。 そういう例は annex の例 No.30（：欠落）にあります。

時には文の証明の処理の過程で１つあるいは複数の補助的補題を justify することが都合の良いことがあります。 ここでレファランス・リストがゼロ・リストの straightforward justification については述べる価値があります。 そのケース(*)では： （訳注： p52 の 11 行の*）

justified-sentence ;

という形を持ちます。 その特別の種類の justification は命題計算の恒真命題（tautology）や関数計算の単純な法則の文についてだけ関係があります。

justification を検証（verify）するシステムの部分は CHECKER と呼ばれます。 恒真命題は CHECKER にとって自明であり justification は必要ありません。 参照がない（with a zero reference）straightforward justificationの例を説明します。

```
Int P = P implies not(Int  P ≠ P & Cl P = P);
```

```
P = Q implies (P is_open iff Q is_open);
for k, l holds k = l or k ≠ l;
```

（ファイル art.lst の例 No.36 を見てください：欠落）。

次の形のスキームによる justification：

justified-sentence **from** *symbol-of-schema (reference-list) ;*

はもう一つの別の straightforward justification です。

リスト - ...というのは任意の表現型なので、レファランス・リストの参照の数はゼロでも良いのです。

レファランス・リストがゼロ・リストであればスキームによる justification は次の形：

justified-sentence **from** *symbol-of-schema ;*

となります。

ファイル art.lst の例４は帰納（induction）のスキームが使われた定理の証明の説明をしています（：欠落）。

上の例は、straightforward justification にだけあてはまります（had to do with）。 しかし、多くの場合定理は証明を必要とします。 そして straightforward justification －特に直接のそれは－ 証明で使われる推論（推論（reasoning）というのは証明中のあるステップのことです）のなかにその応用例を見ることができます。

定理が真であることの証明が直接、あるいはスキーム参照によっても justify できないと、なんらかの証明が行われなければなりません。

定理の内容を書いた後に

proof
…
end;

を書きます。 ここで点々（dots）はもちろんある種の推論（reasoning）のことです。

すべての推論（every reasoning）は一連の推移（successive transition）のひと続き（sequence）で（straightforward justification あるいは証明により）justify されるべきものです。 その観点からの例外は、証明のスケルトンを形成する記述です（仮定、一般化、例示）。 これらは次のセクションで議論しましょう。

証明の中にある justified step はステートメントと呼ばれます。 ステップ

は一体となって証明を形成（combined to form //??from->form）しますが、定理の命題、証明のやり方（例えば、直接証明かあるいは間接証明か）、そして明らかにアーティクルを書く人のイマジネーション、に依存します。

次の位相空間の証明を（Mizar記法への焼き直しではないやり方で）やってみましょう：

位相空間 G の任意の部分集合 A, B について、次の式：

$$\mathrm{Cl}\,(A \cap B) \subseteq \mathrm{Cl}\,A \cap \mathrm{Cl}\,B$$

が成立する。

証明.

位相空間 G の任意の２つの部分集合 A, B について考えてみましょう。

集合の特性（properties）から $A \cap B \subseteq A$ と $A \cap B \subseteq B$ が知られています(//??know->known)。

閉包の演算の次の特性:

$M \subseteq N$ ならば $\mathrm{Cl}\,M \subseteq \mathrm{Cl}\,N$（ここで M, N は位相空間の部分集合です。）

を利用することで

$\mathrm{Cl}\,(A \cap B) \subseteq \mathrm{Cl}\,A$ と $\mathrm{Cl}\,(A \cap B) \subseteq \mathrm{Cl}\,B$

と書くことができます。

かくして、$\mathrm{Cl}\,(A \cap B) \subseteq \mathrm{Cl}\,A \cap \mathrm{Cl}\,B$

証明終わり（quod erat demonstrandum）.

さあ、今度はこれを Mizar記法で書いてみましょう。 今考えている定理の証明を持つ Mizar article は、ご存じのように（//??know->known）、以下の要素から構成されます：

environ
directives of environment
begin
content of theorem
proof
reasoning
end;

問題の定理は、Mizar記法では次のようになります：

```
for A,B being Subset of G holds Cl(A ∩ B) ⊆ Cl A ∩ Cl B
```

式に関する章で述べた事に従うと、上に述べた定理の書き方はいくつかの可能なバージョンのうちの一つに過ぎません。

さて次に当然やるべきこと（proper）、語 **begin** の後に続くテキストの構成

に進みましょう。　article のそれ以外の部分は後で議論します。　その種の証明を書く手順は特にMizarを学び始めた人にとっては重要です。　テキスト本体（the text proper）を書いた後で環境部のどの指令（directive）を語 **environ** と **begin** の間に挿入するべきかを明瞭に知ることができます。　他方、証明のプロセスに使われる相当な時間の間、欠陥のあるまま構築された環境は、より以上に証明を困難にしてしまうことを心にとめておく必要があります、なぜなら *Mizarの手続き（procedure）* においては、証明とは進行中の証明の正当性の検証なのですから、エラーは欠陥のある環境の構文に関係のものが報告されることになるでしょう。

*変数の予約*という構文は、ある位相空間を示す識別子 G に対してのみ使用されます。　残りの識別子の型は必要になった時点で与えられるでしょう。

次のテキスト本文（the next proper）は：

```
reserve G for TopSpace;
 for A,B being Subset of G holds Cl (A ∩ B) ⊆ Cl A ∩ Cl B
      proof
         reasoning
      end;
```

という形を想定（assumes）します。
この時点で、推論（reasoning）の実行が残されています。

前の証明で位相空間 G の任意の二つの部分集合を考えました。　今回も類推しながら進んでみましょう。

語 **proof** の後に：

```
let A,B be Subset of G;
```

を書かないといけません。　この表現はこうして：

A, B を位相空間 G の任意の部分集合とします。

と翻訳することができます。　変数の予約で予約されていませんので **let** A,B **be** Subset **of** G; という表現の中で、変数 A, B の型が設定されねばなりません。*（式のなかで与えられた修飾（qualification）はその式の最後(//??and->end)までのスコープを持ちます。）*

前の証明の模倣を続けて：

```
A ∩ B ⊆ A & A ∩ B ⊆ B;
```

を書きます。　これは疑いなく推論（reasoning）の一ステップです。　(//?? It has

been said earlier that every step of the reasoning.の部分は重複->削除)　以前に推論（reasoning）の全てのステップは justify されなければならない、さもないと CHECKER がエラーNo.4：This reference is not accepted by Checker. を報告してくるだろうと述べました。

今考えているケースでは direct justification で十分（suffice）です；　これは言わば適切な参照の指示を必要としない justification のことを意味しています。

もう一度、定理の内容を持つファイルはサブディレクトリ ¥ABSTR にあること（¥ABSTRは¥MIZAR のサブディレクトリです）、そして拡張子 .abs を持つことに注意してください。　ファイル BOOLE.abs の内容を点検すれば、定理 No.37（すなわちBOOLE:37）に出会うでしょう。　それは任意の集合 X,Y について：

X ∩ Y ⊆ X & X ∩ Y ⊆ Y

であると述べており、この定理の参照することで推論（reasoning）の最初のステップは justify されます。　ステートメント：

Z1:　A ∩ B ⊆ A & A ∩ B ⊆ B **by** BOOLE:37;

を得ます。

次の推論（reasoning）の justify ステップをファイル PRE_TOPC.abs の中にあるはずの定理 No.49 を適用して得ます。　その内容は：

for A, B **being** Subset of 𝒢 **st** A ⊆ B **holds** Cl A ⊆ Cl B

です。　この定理は式：

A ∩ B ⊆ A & A ∩ B ⊆ B

に適用されるはずです。　これが式にラベルを付ける理由です。　そのラベルはこの場合 Z1 という記銘です。　ラベルの識別子は後にコロンが続くということに注意してください。　与えられたラベルを参照することでそのラベルを持つ文を参照します。

推論（reasoning）の第二ステップは次のようになるでしょう：

Z2: Cl (A ∩ B) ⊆ Cl A & Cl (A ∩ B) ⊆ Cl B **by** Z1, PRE_TOPC:49;

証明の後のほうで前提として使われる予定があるので、ご覧のように、この文にはラベルが与えられています。

ご覧のように、定理 PRE_TOPC:49 は２回適用されています、しかし

justification 中では１回しか与えられていません。
注釈：同一の証明ステップで参照が数回指示された場合、それは語 **by** *の後に１回記述するだけで充分です。*

Z ⊆ X & Z ∩ Y **implies** Z ⊆ X ∩ Y （ここでX,Y,Zは任意の集合）を言っている定理 BOOLE:39 を利用して、結論：

thus Cl (A ∩ B) ⊆ Cl A ∩ Cl B **by** Z2, BOOLE:39;

を書き下すことができます。 これは証明の推論（reasoning）の最終ステップです。

語 **thus** は、証明の命題（thesis）の文、あるいはその一部分に先行します。この場合それは証明の命題です。

結局のところ、文の本体（the next proper）は以下のようになります：

```
reserve 𝔊 for TopSpace;
for A,B being Subset of 𝔊 holds Cl (A ∩ B) ⊆ Cl A ∩ Cl B
    proof
      let A,B be Subset of 𝔊;
      Z1: A ∩ B ⊆ A & A ∩ B ⊆ B by BOOLE:37;
      Z2: Cl (A∩B) ⊆ Cl A & Cl (A∩B) ⊆ Cl B by Z1, PRE_TOPC:49;
      thus Cl (A ∩ B) ⊆ Cl A ∩ Cl B by Z2, BOOLE:39;
    end;
```

さて語 **environ** と **begin** の間の環境部に適切な指令を挿入する作業が残っています。 ボキャブラリから始めましょう、指令 **vocabulary** …; から始めます。

上記のテキスト本体は次のシンボルが出現します：

∩、Cl	functorのシンボル
⊆	述語のシンボル
TopSpace, Subset	モードのシンボル

シンボル ∩ はボキャブラリ BOOLE で見つかるはずです。 シンボル Cl と TopSpace は ボキャブラリ TOPCON で導入されますが、モードのシンボル Subset と述語 ⊆ のそれはボキャブラリ HIDDEN の中にあります。 われわれのアーティクルでインクルードされなければならないボキャブラリ指令は：

```
vocabulary BOOLE;
vocabulary TOPCON;
```

です。

指令 **signature** α; はアーティクル α.miz で定義されたフォーマット ― 左側と右側の引数の数、さらに結果の型、functor の引数、あるいはモードの拡張など、に従ってボキャブラリのシンボルの使用を許します（//??on->one）。

われわれの場合、シンボル ∩ は集合の共通部分（intersection）のシンボルとして使用されます。 その共通集合はある位相空間の部分集合の共通集合として取り扱われ、その位相空間はまたその部分集合で、あるいは ― 全く推論の（訳注：記述の）変更なしに ― 複数の集合の通常の（ordinary）共通集合として取り扱われます。 かくしてその使用には指令：

signature SUBSET_1; （部分集合の再定義）

の、あるいは指令

signature PRE_TOPC; （位相空間の部分集合の再定義）

の、あるいは指令

signature BOOLE; （集合の共通集合の定義）

の付加が必要です。

シンボル ∩（訳注：集合では⊂、⊆ですが？）が使われる包含（inclusion）という述語の定義は、アーティクル TARSKI で見つかるはずです。 それをここで使うつもりなら環境の指令のうちの一つは

signature TARSKI;

でなければなりません。

アーティクル PRE_TOPC で集合の閉包、Cl というシンボルが使われ、そしてモード TopSpace の定義が導入されます、それゆえ指令

signature PRE_TOPC;

を付加する必要があります。

証明の中でアーティクル BOOLE と PRE_TOPC の中にある定理を利用するので、以前に環境に書いた指令に指令をもう２つ付け加えなければなりません、すなわち：

theorems BOOLE;
theorems PRE_TOPC;

です。

今考えている定理の証明を含むMizar articleは付録のファイル art.lstの

例５にあります。（：欠落）

注釈： Mizarではラベルのオーバーライドが許されています。　なので複数の同じ推論レベルにある文を同一のラベルでマークしてもエラーになりません。

ある推論のレベルで、そのレベルには他の推論レベルは無いとしますが、複数の文が同じラベルでマークされていると、そのラベルへの参照はそのラベルでマークされた最後の文への参照となります。　推論のレベルとは：

(a) 対応する **proof** と **end;** のペアの間の推論、

(b) 対応する **now** と **end;** のペアの間の推論、

(c) 対応する **scheme** と **;** のペアの間の推論、

を意味します。　すでに閉じられた前の推論のレベルへの参照は許されません。

例

２つの推論のレベルを以下に示します。　α、β、γ、δの記銘はある式を意味します。　示された推論（reasoning）のレベルで、ラベル A: は指定された場所にのみ現れます。

Level of reasoning

```
. . . .
A: α;
. . . .
```

Level of reasoning

```
. . . .
A: β;
. . . .
γ by A;
. . . .
```

```
. . . .
δ by A;
. . . .
```

文γの justification でのラベルへの参照はそのラベル（訳注： A:）でマークされた最後の文、すなわち文 β への参照を意味します。

しかし文 δ の justification ではラベル A： への参照は文 α の参照を意味します。　その場所では２番目のラベル A: とマークされた文への参照は

不可能です。　なぜならその文は既に閉じられた推論（reasoning）のレベルに存在するからです。

長い定理の証明の中では、まえに証明で使われた推論（reasoning）に対応する記述を（訳注：再度）使えると便利です。　それは推論を短くし、より明解に、より可読性を向上させます。　それは証明の中で語 ： **then**、**hence**、そして **thesis** を使用してラベル付きの文を排除することで達成できます。

文βの前に語 **then** を付けることは、β の justification において直前にある文 α を利用できることを意味します。　その場合には文 α をラベルでマークする必要がなくなります。

この justification の方法は *linking* と呼ばれます。　前に行った証明で linking を使ってみましょう。　Z1 と Z2 でラベルされた文の代わりに

```
A ∩ B ⊆ A & A ∩ B ⊆ B by BOOLE:37;
then Cl (A ∩ B) ⊆ Cl A & Cl (A ∩ B) ⊆ Cl B by PRE_TOPC:49;
```

とします。

linking は表現 β が straightforwardly に justify されたステートメントであること、及び α が文であることを要求することを強調しておかなければなりません。　このことは linking の適用に、ある制限を課します。　例えば、linking は **proof** の直後には適用できませんし、集合仮定のどの部分が指示されたかわからないので collective assumption の後にも使用できません。（linkingは）選択のステートメントの後、型変更のステートメントの後、例示（exemplification）の後にも適用できません。　linking は文の直後、選択のステートメントの後で直接 justify したステートメントの直後、そして diffuse statement の後で使用することができます。

先行文が結論の前提（premises）の一つである場合、linking は **thus** を **hence** で置き換えることで指定されます。　結論に先行する文はラベルを廃することができます。　比喩的にいえば（Figuratively speaking）、記述

```
A: α;
thus β by A, other-references;
```

は

```
α;
hence β by other-references;
```

という記述で置き換えて良いのです。

linking が可能と考えられた時、前に議論した定理の証明は：

```
proof
  let A,B be Subset of 𝒢;
    A ∩ B ⊆ A & A ∩ B ⊆ B by BOOLE:37;
   then Cl (A ∩ B) ⊆ Cl A & Cl (A ∩ B) ⊆ Cl B by PRE_TOPC:49;
  thus Cl (A ∩ B) ⊆ Cl A & Cl B by BOOLE:39;
end;
```

という形が想定（assume）されるでしょう。 （art.lst の 例 No.6 を見て下さい：欠落）

結論（すなわち証明の命題あるいはその一部分）、この場合は

```
Cl (A ∩ B) ⊆ Cl A & Cl B
```

という表現です、 は*提示されるべく残されている物*を意味する語 **thesis** で置き換えることもできます。 語 **thesis** は Mizar により式として扱われます。式 **thesis** は証明の中で **proof** と **end;** の間で単独で使われます。

thesis の使用例

- *証明の終止で：*

  … **hence thesis; end;** あるいは … **thus thesis; end;**

（訳注：以下の No. の例は欠落） ファイル art.lst の例 Nos.7, 10, 28, 29 は証明の終わりの部分での式 **thesis** の応用例の説明をしています。 例 No. 7 の最初の内側の証明で、それは命題 P ∪ ΩT = ΩT の証明の中なのですが、**thesis** は式 P ∪ ΩT = ΩT を意味し、二番目の内側では、**thesis** は式 P ∪ ΩT = Pを意味します。

例 No.10 の中で式 **thesis** は２回現れます。 最初のものは式 W is_open & W ⊆ P & x ∈ W を意味します、２番目の場合は証明の終りで、式 x ∈ Int P を意味します。

例 No.28 では **thesis** はその例で証明される文(//??sentenced->sentence)を意味します、それは文

```
T is_connected iff for A being Subset of T st A is_open_closed
    holds A = ∅T or A = ΩT
```

です。 ここで例 No.29 の **thesis** は式 P is_dense を意味します。

- *間接証明の最初で：*

  proof

```
    assume not thesis;
      … ;
    thus contradiction;
  end;
```

このような **thesis** の用法は例 Nos. 19, 22, 23 に見られます。
No.19 の例での **thesis** は証明しようとしている文、つまり式：

for G **st** G is_open **holds** p ∈ G **implies** P ∩ G ≠ ∅.

を意味します。No.22 の例では、それは式 P ≠ Q を意味します。 No.23 の例では、それは式 A ≠ ∅T を意味します。(訳注：以上の No. のつく例は欠落)

- *場合（cases）による証明で：*

```
  proof
      A: now assume  α ;
            … ;
          hence thesis;
        end;
        now assume not  α ;
            … ;
          hence thesis;
        end;
    hence thesis by A;
  end;
```

Ⅳ.２．証明のスケルトン（Skeletons of proofs）

全ての証明は **proof** と **end** の間に含まれる推論（reasoning）で、そのスケルトンからの要素で構成されます。 証明のスケルトンは：

assumption(仮定)、
generalization(一般化)、
conclusion(結論)、
exemplification(例示)

から構成されます。

証明のスケルトンは曖昧さ無く（unambiguously）決定されるのではないことに注意を喚起しておかねばなりません。 その構造は証明されるべき命題の形や、証明のテクニック（technique）に依存します（例えば*直接証明*か*間接証*

明か）。　与えられた命題の証明のスケルトンはその命題の semantic correlates の構造の基盤の上に成立します。　その命題の semantic correlates の構造の正しさを検証するのはシステムの一部(//??述語欠落？->です)。　スケルトンの構造の正当性を検証するシステムのその部分は REASONER と呼ばれます。

文の証明の前に、最初にその文の証明の正しいスケルトンを（すなわち、文の justification を無視して）書くことは賢明な（advisable）ことです。　証明のスケルトンが正しく書かれれば、ナンバー４とマークされたエラーだけが（CHECKER により）レポートされるでしょう。

Ⅳ.２.１.　直接証明（DIRECT PROOFS）

さて、命題が*論理積(conjunction)*、*論理和(disjunction)*、*含意(implication)*、*同値(equivalence)*、*一般化文(general sentence)*、*例示文(existential sentence)*の証明の時によくみられるような（likely）スケルトンを示しましょう。

1. *CONJUNCTION (論理積)* は命題です。 それは α_1 & α_2 の形の表現で、α_1 と α_2 は任意の式です。

この文を*直接法*で証明するつもりならば、証明のスケルトンは下にリストアップされた表現のように構成され：

proof
　……
　thus α_1**;**
　……
　thus α_2**;**
end;

のように語 **proof** と **end** の間に含まれます。
上のドットは証明の残りの（remaining）ステップで置き換えられます。

いうまでもなく（Self-evidently）文 α_1 と α_2 は straightforward justification あるいは proof で証明されなければなりません。　文を証明するスケルトンを構成するコンポーネントの表現はいずれも証明の命題を変化させ（modifies）ます。　上の例では **thus** α_1 という表現が本文中に出現するまでは式

$$\alpha_1 \& \alpha_2$$

が証明の命題でした。　しかし表現 **thus** α_1 は証明の命題 (thesis of the proof)

を変化させ、その表現の（訳注：出現した）後では（訳注： thesis of the proof は）式 α_2 になります。 α_1 と α_2 の justification の順序（sequence）は本質的(essential)です； それは上に示した順番でないといけません。 順番(order)を：

thus α_2;
thus α_1;

に換えると、文 α_2 & α_1 の証明のスケルトンということになります。 しかし

文 α_1 & α_2 と 文 α_2 & α_1

の semantic correlate は異なります。 Mizar にとっては、この２つの文は異なる文なのです。

（訳注：以下の例は欠落）例 `Z8.lst` は justify される文の証明のスケルトンだけを書き下ろしたときの文型のファイルを示しています。 その例の証明で、証明のスケルトンの表現の順序（order）をかえると、`No.51 ―Invalid conclusion―` とマークされたエラーが追加されて報告されるでしょう。そういう状況の説明は例 `Z9.lst` にもあります。

完全な証明はファイル `art.lst` の例 No.7 にあります。（：欠落）

その例の命題の証明で２つの内部的証明があります。 それぞれの証明の最終結論は語 **thesis** でマークされています。 最初の内側の証明でそれは `P ∪ ΩT = ΩT` を、そして２番目では `P ∪ ΩT = T` を示します。

命題が２つ以上の構成成分（constituents）の論理積（conjunction）である時、それぞれの構成要素が真であることを justify するべきです。

例えば、命題

α_1 & α_2 & α_3

に対しては証明のスケルトンは以下のようになるでしょう：

proof
.....
thus α_1;
.....
thus α_2;
.....
thus α_3;
.....

end;

注釈：　命題 α_1 & α_2 & α_3 の証明のスケルトンはまた命題 α_1 &（α_2 & α_3）と命題（α_1 & α_2）& α_3 の証明のスケルトンでもあります。　つまり以上の３つの式は同一の semantic correlate を持つということです。

2.　*IMPLICATION（含意）*は命題です。

含意を証明するには２つの方法があります、直接法と間接法です。

含意

$$\alpha_1 \textbf{ implies } \alpha_2$$

を直接証明するつもりなら、含意の前項の仮定をしなければなりません、次に後項を証明します。　上の文の証明のスケルトンは次のようになるでしょう：

proof
　……
　assume α_1;
　……
　thus α_2;　*（結論）*
end;

証明のスケルトンの一部である仮定は命題を変化させます。　仮定が出現する前には、命題は式

$$\alpha_1 \textbf{ implies } \alpha_2$$

でした。　しかし含意の前項を仮定することは命題を変化させ、仮定出現の後の命題は式 α_2 になります。

例

文の証明のスケルトン

$\Omega\mathfrak{G} = A \cup B \;\&\; A$ is_closed $\&\; B$ is_closed $\&\; A \cap B = \emptyset\mathfrak{G}$
　　implies A,B are_separated

は

proof
　……
　assume $\Omega\mathfrak{G} = A \cup B \;\&\; A$ is_closed $\&\; B$ is_closed $\&\; A \cap B = \emptyset\mathfrak{G}$;
　……
end;

となるでしょう。
（付録　―file Z10.lst を見てください。　：以下例文欠落）
注釈：
　証明のスケルトンを構成する（結論以外の）ステップは justification を必要としません。　残りの証明のステップのうち恒真命題（tautologies）以外は (//??that->than) justify されねばなりません。　これは annex の例 Z10.lst で説明します、そこでは仮定の justification に関連したエラーはありません。

表現

assume ΩG = A ∪ B & A is_closed & B is_closed & A ∩ B = ØG;

は仮定の形式のうちの一つの*シングル・アサンプション*で、*シングル・アサンプション*は下に示す２つの形のうちの１つをとります：

assume *sentence;*

この仮定は、それを linking で参照するときに使われます。　しかし時には仮定の中の文を linking で参照できないことがあります。　その場合は、文はラベルでマークされ、ラベルの識別子が参照のところに書かれねばなりません。その仮定は

assume *identifier-of-label* : *sentence;*

という形を取るでしょう。
　仮定として考える文が論理積（conjunction）の形をしていると、仮定は、記号 & を語 **and** で置き換えて、論理積のすべての構成要素（constituent）にラベルを付けて、*コレクティブ・アサンプション（集合仮定）*という形で記述されます。　従って*シングル・アサンプション（単一仮定）*：

assume ΩG = A∪B & A is_closed & B is_closed & A ∩ B = ØG;

は：

assume that M1: ΩG = A∪B **and** M2: A is_closed **and**
M3:B is_closed **and** M4: A ∩ B = ØG;

と*コレクティブ・アサンプション*の形に書くことができます。　*コレクティブ・アサンプションは*：

assume that　*sequence-of-labelled-sentences* ;

という形をとります。　ラベルつきの文の一続き（sequence）、あるいは複数のラベル付き文は接続子（connective）**and** で一つにリンクされます。

*シングル・アサンプション*を*コレクティブ・アサンプション*に分割すれば、全ての部分的な一つ一つの仮定を個別に参照できます。

上に挙げた例の仮定はまた：

```
assume x: Ω𝔊 = A ∪ B & A is_closed & B is_closed;
assume xx: A ∩ B = Ø𝔊;
```

という形を持つこともあります。
α & β & γ と α &（β & γ）と（α & β）& γ の semantic correlates が同じであることから以下の仮定：

```
assume y: (Ω𝔊 = A ∪ B & A is_closed) & B is_closed;
assume yy: A ∩ B = Ø𝔊;
```

もまた正しいのです。 同じことが次の：

```
assume z: Ω𝔊 = A ∪ B & (A is_closed & B is_closed);
assume zz: A ∩ B = Ø𝔊;
```

にもあてはまります。 最後の２つの形式の仮定は読み易くありません、余計なカッコを省いた記述の使用のほうが良いとされる理由です。 しかし最後の２つの形式は多様な記述が可能であることを示すために挙げました。

（●）Φ_1 & Φ_2 **implies** Φ_3 と Φ_1 **implies** (Φ_2 & Φ_3)

という形式の式は同一の semantic correlate を持ちます。
Φ_1、Φ_2、Φ_3 は任意の式です。 それらが含意や同値の式であれば（●）につづく式でカッコは適切な場所に置かれるはずです。 同じことがこれから議論する下の式についても適用されます。 式 Φ_1 & Φ_2 **implies** Φ_3 の semantic correlates は次のように：

not ((⌜Φ_1⌝ & ⌜Φ_2⌝) & ⌜**not** Φ_3⌝)

の形を持ちます。 （ここで記銘 ⌜Φ_1⌝ は式 Φ_1 の semantic correlate を表します； 類推により、同じことを残りの場合にも適用してください。）
式 Φ_1 **implies** (Φ_2 & Φ_3) の semantic correlate がどのように形成されるかを示しましょう。 その式は：

not (Φ_1 & **not** (Φ_2 **implies** Φ_3))

と同値の式に書くこともできます。 この式は含意の semantic form を使用して：

not (Φ_1 & **not not** (⌜Φ_2⌝ **&** ⌜**not** Φ_3⌝))

と同値の式に書きかえることもできます。　次に否定（negation）は一種の退縮（involution）であるという事実を利用して：

not (Φ_1 & (⌜Φ_2⌝ **&** ⌜**not** Φ_3⌝))

とします。　論理積（conjunction）は結合律を満たす（associative）ので、上の式の semantic correlate は：

not ((⌜Φ_1⌝ & ⌜Φ_2⌝) & ⌜**not** Φ_3⌝)

と書けます。　このようにして、得られた形はまた、式：

(Φ_1 & Φ_2) **implies** Φ_3

の semantic form でもあります。

証明実行中の式は：

α_1 & α_2 & α_3 & α_4

の形の前項をもちます。　ここで

α_1 は　Ω𝔊 = A ∪ B
α_2 は　A is_closed
α_3 は　B is_closed
α_4 は　A ∩ B = Ø𝔊

を表します。　含意の後項は式 A,B are_separated なのですが、γ で表しましょう。

システムは式を読み込んだ時適切な位置にカッコを付け加え，変形して：

((α_1 & α_2) & α_3) & α_4

の形にします。　今、((α_1 & α_2) & α_3) を β で表しましょう。　前の式に β を代入して

β & α_4

を得ます。

以前の解析から式：

β & α_4 **implies** γ　と　β **implies** (α_4 **implies** γ)

は同一です。　こうして証明の命題のスケルトン

β & α_4 **implies** γ

は証明の命題のスケルトン

β **implies** (α_4 **implies** γ)

と同じです。 文

```
ΩG = A ∪ B & A is_closed & B is_closed & A ∩ B = ØG implies
A, B are_separated
```

はシステムにより文

```
ΩG = A ∪ B & A is_closed & B is_closed implies (A ∩ B = ØG
implies A,B are_separated)
```

と同じであると処理されるので、命題の証明のスケルトン

```
ΩG = A ∪ B & A is_closed & B is_closed & A ∩ B = ØG
    implies A, B are_separated
```

は：

```
proof
    ......
      assume q: ΩG = A ∪ B & A is_closed & B is_closed;
(今命題は： A ∩ B = ØG implies A,B are_separated です)
      ......
      assume p: A ∩ B = ØG;
(今命題は： A,B are_separated です)
      ......
      thus A,B are_separated;
      ......
end;
```

という形になります。

この例で議論した命題の証明はファイル art.lst の例 No.9 に示されています（：欠落）。

注釈：文 α & β **implies** γ *の証明のスケルトンは下に示すようになります：*

```
  proof
    ......
      assume α
```

```
        .....
        assume β
        .....
        thus γ
      .....
    end;
```

これとは逆に、それ（訳注：上記スケルトン）は文

β & α **implies** γ

の証明のスケルトンにはなりません。　なぜなら、そのケースでは異なった仮定の順序が必要だからです。

3. *同値（EQUIVALENCE）* α **iff** β を証明するには２つの含意：
α **implies** β と β **implies** α を証明しなければなりません。

α **iff** β と（α **implies** β）&（β **implies** α）は同一の semantic correlates を持つので、同値（equivalence）の証明のスケルトンは論理積（conjunction）の命題の証明のスケルトンに包括（subsumed）されます。

同値の証明のスケルトンは：

```
α iff β
proof
    .....
    thus α implies β;
        (今命題は：α implies β です)
    .....
    thus β implies α;      (原文は　thus α implies β; です)
    .....
end;
```

となります。

Mizarにとって論理積（conjunction）の交換律（commutativity）は自明ではないので含意が示されるのは上記の順番でないといけません。　ファイル `art.lst` の例 No.10 は同等（equivalence）の証明を示します（：以下 No. を持つ例文は欠落）。　その証明のスケルトンは上と同じです。

同等の証明 α **iff** β のスケルトンは以下のようになります：

```
proof
```

.....
thus α **implies** β;
*(今命題は：*β **implies** α *です)*
.....
assume β ;
*(今命題は：*α *です)*
.....
thus α;
.....
end;

annex の例 No.11 に例 No.10 から取った命題の証明があります。 その証明は例 No.10 とは異なる方法で、つまり異なる証明のスケルトンで行われています。 それは上のような形です。

同値（命題）の justification はファイル `art.lst` の例 No.30 で見ることができ(以上 No.を持つ例文は欠落)、同値(命題)は**by**により straightforwardly に justify されます。 参照のリストは以前に証明された２つの対応する含意命題のラベルからなります。

4. *論理和（DISJUNCTION）の命題（THESIS）*を証明するときは、 システムが同じ方法で処理する、対になるもう一つの文を心に留めておく価値があります。

それは文

α **or** β と **not** α **implies** β

のことです。 これらの文は論理和（disjunction）の証明を持ちます。 論理和の最初の構成成分（α）の否定を仮定し、２番目の構成成分を証明するのが便利です。 逆に２番目の構成成分（β）の否定を仮定して、最初の（α）を証明して命題 β **or** α を証明するべきでしょうか？
しかし（訳注：そうではなく）文

α **or** β と β **or** α

は異なる semantic correlate を持つのです。

命題が３つの構成要素の論理和であれば証明は、はじめの２つの構成要素の否定を仮定し、３番目を証明せねばなりません。 これは以下の例で行われています。

例

```
k < n  or  k = n  or  n < k
proof
    assume A: not k < n & k <> n;
(最初の論理和の２つの構成要素を仮定し、いま命題は式 n < k です)
    then not k ≤ n by NAT_1:30;
    then n ≤ k by NAT_1:14;
    hence n < k by REAL_1:57, A;
end;
```

ファイル art.lst の例 No.12 を見てください。（: 欠落）

5. 命題が*一般化文（GENERAL SENTENCE）*の形式。

この場合は証明のスケルトンの構文は一般化（generalization）で始めなければなりません。　一般化は、例えば、一般化文の証明と一般のものとして提示できるもの（can be presented as general ones）の証明のなかで用いられます。他の一般化は：

－*diffuse statement,*
－*definition*

に出現します。

一般的には(//??General->Generally)、一般化はあるオブジェクトの固定(fix)が目的です。　したがって、その proof のレベルに複数の定数を導入します。

一般化は：

let *variables-qualified-implicitly* **;**

という形式をしています。

修飾された変数（qualified variables）のリストの多様な形態（diversified forms）という観点から、一般化は下に示す表現の形のうちの一つをとるでしょう：

(a)　**let** *identifiers-of-variables* **;**

例：　**let** x;　　**let** A,B;

この種の一般化では、上の例で見られるように、そこに現れる変数の型は指定されません。　このことは、それらの変数の識別子は変数の予約の時に与えられたそれぞれの型を持っていることを意味します。

(b) **let** *variables-qualified-explicitly* ;

この形式の一般化は上記の（preceding one）、現れた変数の型が指定されるものとは異なります。
例：

```
let x,y be Any;
let P,Q be (Subset of G)、p be Point of G;
let a,b be Subset of the carrier of Y;
let a be Subset--Family of the carrier of Y;
（ここで Y はある位相構造体です）
```

be の代わりに **being** を使うこともできます。　しかし慣例では **be** は **let** … という構文のなかで使うことになっています。

(c) **let** *variables-qualified-explicitly, variables-qualified-implicitly;*

一般的には、この一般化は前の２つの組み合わせ（combination）です。
例：

```
let A be (Subset of G), x;
let P1, P2 be (Subset of G), p,q be (Point of G), x,y,z;
```

（●）**for** *lists-of-qualified-variables* **holds** Φ_1 **implies** Φ_2

の形式の文はシステムにより文

（●●）**for** *lists-of qualified-variables* **st** Φ_1 **holds** Φ_2

と同じものとして扱われるのが知られています。　証明しようとする文の命題が（●）あるいは（●●）の文と同じ形を持つならば、一般化は以下の：

let *lists-of-qualified-variables* **such that** *conditions* ;

のように書かれます。
　この種の一般化における条件は単一ラベル文、あるいは語 **and** でリンクされた複数ラベル文として書かれなければなりません。　（●）あるいは（●●）でマークされた式に対し条件は、例えば(//??instant->instance)：

W1：Φ_1;　あるいは　W1：Φ_1 **and** W2： **not** Φ_2;

と記述されます。
　証明における一般化の使用は今から議論しようとしている一般化文の証明のスケルトンの議論で明らかになる（visible）でしょう。　さしあたり一般化は

全称量化子（universal quantifier）の命題の中の*切り落とし（cut down）*であるとだけ言っておきます。　一般化文の証明は例を参照しながら議論することにします。

下で示される個々の例において、対応する変数の予約と証明が必要な文の内容は直接的で、証明のスケルトンの構文（construction）にとって重要です。　間接証明ではスケルトンは違って見えます。　それについては後で議論しましょう。

お話した（announced）例をここに示します：

例 1

reserve 𝔊 **for** TopSpace, x **for** Any, P **for** Subset **of** 𝔊;
for x **holds** x ∈ Fr P **implies** x ∈(Cl (P^c) ∩ P) ∪ (Cl P ¥ P);
proof
.....
(この時点で命題は一般化ステートメントなので、証明のスケルトンの構文は一般化から始まります)
let x;
(識別子 x の型は変数の予約で与えられているので再度与える必要はありません。　一般化は最初の命題の全称量化子の切り落とし（cutting down）という結果を引き起こし、それによって命題は変化（modified）します。 今、命題は含意の形式を持ちます。　含意（implication）を直接証明するときは、その前項（antecedent）を仮定し、その後項（consequent）を証明します。　さらに一般化は定数 x をその proof のレベルに導入します）。
.....
assume x ∈ Fr P;
(今度は、式 x ∈ (Cl (P^c)∩P) ∪ (Cl P ¥ P) が命題となります。)
.....
thus x ∈ (Cl (P^c)∩P) ∪ (Cl P ¥ P); （最終結論）
end;

一般化の中で、そして、それゆえ証明全体の中でも、x 以外の識別子を使おうと思えば使えます、なぜなら一般化は語 **for** の後の量化式に現れる変数の識別子の型に適用されるからです。　大事な点は一般化の識別子の型は量化式の中の **for** に続く識別子の型と一致させなければならないことです。

例えば：

for x **being** T **holds** Φ(x)

(ここで T はある型です)

という形の量化式があるとします。　一般化は：

let y **be** T;

のようになります。

変数の予約のなかで識別子 y が T 以外の型として予約されているとき、あるいは構文中で型がまったく考慮に入ってなかったとすると、一般化の中で適切な型を指定しなければなりません。　両方の場合で一般化による変数の型の導入は、その推論（reasoning）のレベル、すなわちその変数が導入されたレベルが終了するまで有効です。　しかし識別子 y がすでに型 T として予約されていると一般化は以下のようになります：

let y;

注釈：　一般化で導入された変数は他の一般化、選択のステートメント、型変更のステートメント、例示、存在仮定や変数のローカル定義などによりオーバーライドされることがあります。

表現 **assume** x ∈ Fr P; は仮定の形の一つで、シングル・アサンプション（single assumption）です。

いま証明中の文はMizarにとって文：

for x st x ∈ Fr P **holds** x ∈ (Cl (P^c)∩P) ∪ (Cl P ¥ P);

と同じ意味を持ちます。　というわけで純化量化子（purified quantifier）を持つ命題の場合の一般化の構造についての情報に従って、証明のスケルトンは：

let x;
assume x ∈ Fr P;　} **let** x **such that** A: x ∈ Fr P;

と短縮されます。　最初の例で議論した文の証明は annex のファイル art.lst、No.13 以下に示してあります（：欠落）。

さて証明のスケルトンの構文を説明するもう一つの例です。

例２

reserve 𝒢 **for** TopSpace, P **for** Subset **of** 𝒢;
P ⊆ Cl P;

この文の定義の拡張（definitional expansion）は次の形式を持ちます：

for x **being** Any **holds** x ∈ P **implies** x ∈ Cl P

文P ⊆ Cl Pの証明のスケルトンは文の定義の拡張（definitional expansion）の証明のスケルトンでもあります。　これは定義の **definitions** TARSKI; 指令を環境に付け加えることで保証（guaranteed）されます。

これは文 P ⊆ Cl P; の証明のスケルトンです：

proof

.....

（この時点で一般化文に拡張されてもよさそうな文は命題なので、一般化が証明のスケルトンの最初の要素になるでしょう）

let x **be** Any;

（今、命題は含意なので、証明のスケルトンは依然として含意の前項を仮定することから構成されます。）

.....

assume x ∈ P;

（今、それは式　x ∈ Cl P です。）

.....

thus x ∈ Cl P;

.....

end;

完全な証明は annex の例 No.14 でみることができます（：欠落）。

例３

reserve 𝔊 **for** TopSpace, P,Q **for** Subset **of** 𝔊;

P is_dense **implies for** Q **holds** Q≠∅ & Q is_open **implies** P ∩ Q ≠ ∅

proof

.....

(今、命題は含意なので、命題の前項の仮定が証明のスケルトンの最初の要素になります)

assume P is_dense;

（この時点で命題は一般化文です、これは一般化が次の要素になるということを意味します。）

.....

let Q;

（今、命題は含意なので、命題の前項の仮定をします。）

```
        ......
        assume Q ≠ Ø & Q is_open;
```
(今の命題は式P ∩ Q ≠ Øです。)
```
        ......
        thus P ∩ Q ≠ Ø;  (最終結論)
        ......
    end;
```

単一仮定（single assumption）

```
        assume Q ≠ Ø & Q is_open;
```

は集合仮定（collective assumption）:

```
        assume that M1: Q ≠ Ø and M2: Q is_open;
```

あるいは２つのシングル・アサンプション：

```
        assume a: Q ≠ Ø;
        ......
        assume b: Q is_open;
```

として等価に（equivalently）記述できます。

考えている文の証明のスケルトンは以下のようになります：

```
    proof
        ......
```
(今、命題は含意なので命題の前項の仮定が証明のスケルトンの最初の要素になります)
```
          assume P is_dense;
```
（この時点で一般化文が命題で、これは一般化が次の要素になるということを意味します。）
```
        ......
        let Q such that Z1: Q ≠ Ø and Z2: Q is_open;
```
(式 P ∩ Q ≠ Ø;が命題です。)
```
        ......
        thus P ∩ Q ≠ Ø;  (最終結論)
        ......
    end;
```

（annex　―　例No.15を見てください。：欠落）

例４

reserve 𝔊 **for** TopSpace;
for 𝔖 **being** SubSpace **of** 𝔊 **for** A **being** Subset **of** 𝔖
　holds A **is** Subset **of** 𝔊

証明のスケルトン：

proof
　　.....
（今、証明されるべき一般化文が命題です。）
　　let 𝔖 **be** SubSpace **of** 𝔊;
（一般化文：
　　for A **being** Subset **of** 𝔖 **holds** A **is** Subset **of** 𝔊;
が、今の命題です。）
　　.....
　　let A **be** Subset **of** 𝔖;
*（式*A **is** Subset **of** 𝔊 *が命題です。）*
　　.....
　　thus A **is** Subset **of** 𝔊;
　　.....
end;

上の証明の一般化はもっと短く記述できます。　すなわち：

let 𝔖 **being** (SubSpace **of** 𝔊), A **being** Subset **of** 𝔖;

です。　これは文

for x **for** y **holds** α　と　**for** x,y **holds** α

が同一の semantic correlate を持つという事実によります。
この例の命題の証明は annex の例 No.16 にあります。(：欠落)

6.　命題が存在文（*EXISTENTIAL SENTENCE*）、すなわち
　　ex *list-of-qualified-variables* **st** *formula* という形の文のとき。
式

ex x **being** T **st** Φ(x)

が命題だとしましょう。

この命題の証明は条件 Φ(x) を満たす T 型のオブジェクトを指示する（indicating）ことにあります。

そのため **take** ..... という*例示（exemplification）*と呼ばれる構文を利用します。 この構文は、一般化、仮定、そして結論の場合を除いて、証明の命題を変化させます。 一般化（generalization）が命題の中の全称量化子の*切り落とし（cutting down）*という結果になる一方で、例示（exemplification）は命題の中の存在量化子（existential quantifier）を*切り落とし（cuts down）*ます。同等化（equalization）を伴う例示は、その proof のレベルに定数を導入します。 その定数は、他の例示、一般化、選択のステートメント、型変更のステートメント、存在仮定（existential assumption）、変数のローカル定義などでオーバーライドされなければ、導入された時点からその推論のレベルの最後までアクセス可能です。

例として定理：

（●） `x ∈ Int P` **iff ex** `Q` **st** `Q is_open & Q ⊆ P & x ∈ Q`

を考えてみましょう。 この定理の証明は２つの含意の justification から成ります。 その最初のほうの証明のスケルトンを書きましょう。

```
X ∈ Int P implies ex Q st Q is_open & Q ⊆ P & x ∈ Q
proof
    .....
```

（含意が命題なので命題の前項の仮定を行います。）

```
        assume x ∈ Int P;
```

（今、命題は存在文です。 その文は `Q` *equal* `Int P` *を満たすことに注意してください。）*

```
        .....
        take Q = Int P;
```

（この時点で式 `Q is_open & Q ⊆ P & x ∈ Q` *が命題です。 構文* ***take*** *…により探しているオブジェクトを指示します。 それが語* ***st*** *の後に述べられている条件、つまり考えている命題ですが、を満足しているかどうかを検証しなければなりません。 もちろん、例示の中の識別子* `Q` *は、今後の証明でもそうですが、任意の識別子で置き換えてかまいません。）*

```
        .....
        thus Q is_open & Q ⊆ P & x ∈ Q
        .....
```

end;

完全な証明は例 No.10 にあります。(: 欠落)
上で証明された命題にも、構文 **take** … の他のバリエーションがあります ;
表現

take Q = Int P;

は **take** Int P; で置き換えても良いでしょう。 その場合の証明は annex ― 例 No.11 にあります (: 欠落)。
(●) でマークした命題の一部分で、そのほかの含意の証明を例 No.10 と No.11 に示します。 それぞれの例で証明は異なる方法で行われています。(: 欠落)
ここに文

ex a **st** α(x) の証明の２つのスケルトンの例を提示します。

(i)
proof
.....
take y = τ;
.....
thus α(y);
.....
end;

(ii)
proof
.....
take τ;
.....
thus α(τ);
.....
end;

(今 τ は (訳注 : (i) と (ii) で) 対応する項です、そして y は任意の識別子です。 任意の識別子を y に代入できます。)

証明の文のなかには複数の項の等価 (equalization) から構成される例示を持つものもあります。 その場合、一つ一つはコンマで (//??be->by) 区切られていなければなりません。
文

ex x **ex** y **st** α(x,y)

を命題としましょう。　証明のスケルトンは次のようになる（might be）でしょう：

```
proof
    ......
    take x;
    ......
    take y;
    ......
    thus α(x,y);
    ......
end;
```

例示を２回適用しますが、一回しか実行されません。　証明のスケルトンは：

```
proof
    ......
    take x,y;
    ......
    thus α(x,y);
    ......
end;
```

となります。　表現 **take** x,y; もまた命題

ex x **ex** y **st** α(x,y)

の例示です。　しかし文

ex x **ex** y **st** α(x,y)　　と　　**ex** x,y **st** α(x,y)

はシステムにおいても同様に使用可能（ready）となっています。

注釈：　（例 No.7 例 No.8 は欠落）

証明に何の貢献もしないステートメントや **not contradiction** *すなわち VERUM（ラテン語;「真正の」）のような統語的関連（sntactic correlate->syntactic correlate）の証明の中へ追加（adding）はエラーになりません。　例えば annex の例 No.7 から取った命題の証明で付加的な結論（additional conclusion）を書きます、それで証明は例 No.8 の形をとります。　余計な* ***thus thesis*** *が加えられてもエラーは起きません。　その場合の* **thesis** *は式* **not contradiction** *(VERUM)です。　その証明の式を semantic correlate に変換する処理で*

VERUM としての **not contradiction** *は無視されます。　同様に* **assume not contradiction** *の追加は上と同じ理由でエラーになりません。*

Ⅳ.２.２.　間接証明（INDIRECT PROOFS）

これまでは直接証明の議論でした。　しかし Mizar では間接証明を行うこともできます。　その場合の証明のスケルトンはどのようになるのでしょうか？

文 α を間接証明しようとします、まずその文の否定を仮定します、そして矛盾（contradiction）の点に到達するまで証明を実行します。　α のための証明のスケルトンは以下のようになるでしょう（might be）：

proof
　……
　assume not α **;**
　……
　thus contradiction;
　……
end;

さもなければ以下のようです。

proof
　……
　assume not α **;**
　……
　thus thesis;
　……
end;
（この場合 **thesis** は式 **contradiction** を意味します）

そのほうが便利ならば、式 **not** α はすでに否定の形にしておいた（already negated）文 α で置き換えることもできます。　Mizar はそれについて無関心です。

語 **contradiction** は論理定数、*偽（falsehood）*を表します。　語 **not contradiction** あるいは VERUM が論理定数、*真（truth）*を意味することは自明です。　語 **contradiction** は Mizar により式として扱われます。　それは間接証明の最後に現れるだけではありません。
他に現われる場所は　：
―フレンケル項（Frænkel's term）の中で：

例えば　　{k + 1: **not contradiction**}

間接証明は含意のときは頻繁に使われ、それは命題：

α **implies** β

です。　間接証明の中では含意の前項、および後項の否定を仮定しなければなりません。　仮定は下に示すように、単一仮定あるいは集合仮定です：

assume α **;**
assume not β **;**

あるいは

assume α **&** **not** β **;**

あるいは

assume that S1: α **and** S2: **not** β **;**

証明は矛盾点に到達する点まで続けられ、その点は **thus contradiction** というステートメントが justify されることで明らかになります。

例 1

P is_open & P is_nowheredense **implies** P = Ø
proof
.....

という文の証明のスケルトンを書いてみましょう。
（今証明している含意が命題です。　間接的に含意を証明するときは、その前項と後項の否定を仮定します。　この仮定は集合仮定の形で記述されます。）

assume that Z1:P is_open **and** Z2: P is_nowheredense **and** Z3: P ≠ Ø**;**

（式 P ≠ Ø は間接証明の仮定です。　証明のステップがすすむと矛盾（contradiction）を引き起こし（yields）ます。）

.....
thus contradiction;
.....
end;

annex の例 No.17 は含意命題の間接証明の説明です。（：欠落）

例２

間接証明のスケルトンを書いてみましょう：

(**for** G **st** G is_open **holds** p ∈ G **implies** P ∩ G ≠ Ø) **implies** p∈Cl P
proof

.....

assume A0: **for** G **st** G is_open **holds** p ∈ G **implies** P ∩ G ≠ Ø;
（今、命題は p ∈ Cl P *です。　問題の含意は間接証明なので、含意の後項の否定を仮定しなければなりません。）*

.....

assume not p ∈ Cl P;

.....

thus contradiction;

.....

end;

完全な証明は例 No.18 にあります。(：欠落)

表現 **not** p ∈ Cl P は等価な表現 **not thesis** と置き換えてもよいでしょう、ここで式 **thesis** は式 p ∈ Cl P を表します。　それから間接証明のスケルトンは以下のようになります：

proof

.....

assume A0: **for** G **st** G is_open **holds** p ∈ G **implies** P ∩ G ≠ Ø;

.....

assume not thesis;

.....

thus contradiction;

.....

end;

ここに例 No.2 からの、もう一つの命題証明のスケルトンがあります

proof

.....

assume A0: **not thesis** ;
（ここの式 ***thesis*** *は証明される文を表します。　更なる証明のステップは必ず*

矛盾（contradiction）を引き起こします。）

……

thus contradiction;

……

end;

この形式の記述に対しての、システムによる正当性証明のチェックには、前の記述（訳注：直接証明）に比べて長い時間が必要です。 例 No.19 は例 No.18 からの命題の証明を示しますが、命題の証明のスケルトンは上に示したのと同じ形を持ちます。（：欠落）

例３

以下の文を証明するスケルトンを書いてみましょう：

```
A is_a_component_of G & B is_a_component_of G
    implies A = B or A,B are_separated
proof
```

……

（証明されつつある含意が命題です。 直接証明をしましょう。 その前項を仮定し後項を証明するのです。）

```
assume Z1: A is_a_component_of G & B is_a_component_of G;
```

（今、論理和 `A = B or A, B are_separated` *が命題です。 まずその論理和の最初の構成要素の否定を仮定し、二番目が真であることを証明します。*

……

```
assume Z2: A ≠ B;
```

（今、文 `A,B are_separated` *が命題です。 この文は間接的に証明される予定ですから、その否定を仮定します。）*

……

```
assume Z3: not A,B are_separated;
```

*（証明が次のステップにすすめば、はかならず矛盾（*contradiction*）を引き起こします。）*

……

thus contradiction;

……

end;

この命題の証明は annex の例 No.20 にあるはずです。　他の間接証明は例 Nos.21, 22, 23, 24 に示します。（: 欠落）

一連の（successive）Mizar 文の証明において構文 **consider** … が使われることがあります。　証明でのその構文の役割は、その proof のレベルに定数を導入することです。

構文 **consider** … が呼ばれたとき、*選択のステートメント（statement of choice）*は以下の形式になります：

(1)　**consider** *list-of-qualified-variables* ;

例えば：

```
consider x,y;
consider A being Subset of G, a being Any;
consider V being set, P,Q;
```

(2)　**consider** *list-of qualified variables* **such that** conditions　*justification*;

条件（condition）は単一ラベル、あるいは複数ラベルで語 **and** により一体にリンクされた文の形です。　Justification は **by** の時は **by**、あるいはスキームのときには **from** により行います。　条件中に現れる単一（あるいは複数）のラベル付文は選択のステートメントの後では linking が許されないという理由によります。　というわけで選択のステートメントの中にある条件は justification を要求しません。　そこで選択のステートメントは次のような形式をとるでしょう：

(3)　**consider** *list-of-qualified-variables* **such that** conditions;

選択のステートメントは以下の２つの時、この形式をとります：

- 選択のステートメントの justification では、linking で直前の文のみ（solely）を参照します。
- 条件は CHECKER により justification なしに受け入れられ、それはつまり命題計算（propositional calculus）の恒真命題（tautology）、あるいは関数計算（functional calculus）の単純な法則にかかわりがある（have to do with）ものについてです。

下の例は選択のステートメントの応用例の説明です。

例４

ここに文の証明があります

```
P is_boundary iff (for Q st Q ⊆ P & Q is_open holds Q = ∅)
```

proof

thus P is_boundary **implies** (**for** Q **st** Q⊆P & Q is_open **holds** Q = Ø)

proof

(今証明中の命題は含意です。　その前項の仮定をおこないます。)

assume P is_boundary;

(今、一般化文 **for** Q **st** Q ⊆ P & Q is_open **holds** Q = Ø *が命題です。)*

then P: P^c is_dense **by** TOPS_1:83;

let Q;

(今、間接証明される予定の含意 Q ⊆ P & Q is_open **implies** Q = Ø *が命題です。)*

assume that P1: Q ⊆ P **and** P2: Q is_open **and** P3: Q ≠ Ø;

(更なる証明のステップは必ず矛盾（contradiction）を引き起こすはずです)

P^c ∩ Q ≠ Ø **by** TOPS_1:80, P3, P2, P;

then Q ∩ P^c ≠ Ø **by** BOOLE:66;

hence contradiction by P1, TOPS_1:20;

end;

*(今、次の含意：//??(を追加-> (***for** Q **st** Q ⊆ P & Q is_open **holds** Q = Ø) **implies** P is_boundary *が命題です)*

thus (**for** Q **st** Q ⊆ P & Q is_open **holds** Q = Ø) **implies** P is_boundary

proof

(今、上の含意が命題です)

assume K: **for** Q **st** Q ⊆ P & Q is_open **holds** Q = Ø;

(今、間接証明すべき式 P is_boundary *が命題です。)*

assume not P is_boundary;

(間接証明の仮定。　さらなる証明のステップは必ず矛盾（contradiction）を引き起こします。)

then not P^c is_dense **by** TOPS_1:83;

then consider C **being** Subset **of** $\mathbb{G}$ **such that** Q: C ≠ Ø

and Q1: C is_open **and** Q2: P^c ∩ C = Ø **by** TOPS_1:80;

C ∩ P^c = Ø **by** Q2, BOOLE:66;

then C ⊆ P **by** TOPS_1:20;

hence contradiction by K, Q, Q1;

end;

end;

（annex の No.21 を見て下さい。　：欠落）

間接証明の仮定の直後に現れる２つのステートメントを解析してみましょう。

最初のステートメントは **not** Pᶜ is_dense; です。　定理 TOPS_1:80 は稠密（ちゅうみつ）な集合（dense set）の特性を定式化しています。

P is_dense **iff**(**for** Q **st** Q ≠ Ø & Q is_open **holds** P ∩ Q ≠ Ø)

しかしステートメント **not** Pᶜ is_dense; は集合 P の補集合は稠密ではないと言っています。　それから命題 TOPS_1:80 を利用できるように記述を加えて：

(1)　**ex** Q **st** Q ≠ Ø & Q is_open & Pᶜ ∩ Q = Ø;

を推論（infer）することができます。　上の条件を満たすオブジェクトが存在するので、更なる解析では（訳注：識別子は）恣意的（arbitrary）なもので良いのです。　しかしその型は(1)式の識別子 Q に合致させねばなりません。　つまり Subset of 𝔊 型でなければなりません。　こうして：

consider C **being** Subset of 𝔊 **such that** Q: C≠Ø;
　and Q1: C is_open **and** Q2: Pᶜ ∩ Q = Ø;

と書くことになります。　選択のステートメントの後で識別子 C の型は Subset of 𝔊 に設定されます。　その識別子が変数の予約で既に他の型に予約されているなら、選択のステートメントはその型をオーバーライドします。　選択のステートメントで導入された定数は、導入時点から推論（reasoning）のレベルの終了までアクセス可能です(//??in accessible->is accessible)、すなわち、そのレベルとはその定数が導入されたレベルです。

注釈：　選択のステートメントで導入された定数は、一般化、他の選択のステートメント、型変更のステートメント、例示、存在仮定、そして変数のローカル定義などによりオーバーライドされることがあります。

選択のステートメントの応用例：

(1)　**ex** x **st** x ∈ X ∩ Y;
　　then consider x **such that** Z: x ∈ X ∩ Y;

(2)　X meets Y;
　　then consider x **such that** a: x ∈ X ∩ Y **and** b: x ∈ Y **by** BOOLE:15;

ここに定理 BOOLE:15：
　X meets Y **iff ex** x **st** x ∈ X & X ∈ Y;
があります。

(3)　X ≠ Ø;
　　then consider x **such that** c: x ∈ X **by** BOOLE:1;

ここに定理 BOOLE:1:
　　X = Ø **iff not ex** x **st** x ∈ X;
があります。

証明における選択のステートメントの応用例は annex の例 Nos. 11, 15, 21, 22, 23, 25 を使って説明します。　さらに例 No.35 は **proof** の外で、すなわち **proof** と **end;** の間に含まれる推論の外での構文の応用例を示します。(：欠落)

Ⅳ.２.３. 新しい Mizar の構文について（ON A NEW MIZAR CONSTRUCTION）

前項が存在の文である含意命題を命題としましょう。　従ってそれは次の形式の文になります：

(**ex** x **being** T **st** α(x)) **implies** β

その命題の証明は次の形になるでしょう(//??may be take->may take あるいは may be)

proof
　　assume A: **ex** x **being** T **st** α(x);
　　consider y **being** T **such that** Z: α(y) **by** A;
(*or* **then consider** y **being** T **such that** Z: α(y);)
　　..... (*β の proof*)
end;

選択のステートメントの識別子は恣意的に選ぶことができます、しかしそれだから、型は仮定の中の識別子 x の型、すなわち T の型と一致させないといけません。

仮定と選択のステートメントは、その場合、*存在仮定 (existential assumption)* で置き換えられることがあります：

given x **being** T **such that** Z: α(x);

命題が前項に存在のステートメントをもつ含意命題ならば、(構文 **assume** … による) あるオブジェクトの存在の仮定と、その仮定により justify される選択のステートメントを存在仮定（existential assumption）で置き換えても良い

でしょう。
存在仮定（existential assumption）は以下の形となります。

(a) **given** *list-of-qualified-variables;*
(b) **given** *list-of-qualified-variables* **such that** *conditions;*

条件は単一ラベルの文、あるいは語 **and** でリンクされて一体になった複数ラベルの文になるでしょう。
存在仮定（existential assumption）で導入された定数の（訳注：アクセス可能）範囲（range）は選択のステートメントで導入された定数の範囲と同じです。

注釈：
存在仮定（existential assumption）の後では、リンキングは許されません。
存在仮定の応用例については例で説明します。

例 1

```
reserve G for TopSpace, P,Q for (Subset of G), x for Any;
(ex Q st Q is_open & Q ⊆ P & x ∈ Q) implies x ∈ Int P
    proof
       assume ex Q st Q is_open & Q ⊆ P & x ∈ Q;
       then consider Q such that Z1: Q is open and
       Z2: Q ⊆ P and Z3: x ∈ Q;
```

（証明中の含意の前項の仮定がおこなわれていて、適切な選択のステートメントがおこなわれています。　今、式 x ∈ Int P が命題です。　証明の残りのステップを以下に示します。）

```
        P^c ⊆ Q^c by TOPS_1:15,Z2;
        then Z4: Cl(P^c)⊆(Q^c) by TOPS_1:25;
        Q^c is_closed by Z1, TOPS_1:30;
        then Cl(Q^c) = Q^c by PRE_TOPC:52;
        then Cl(Q^c) ⊆ Q^c by Z4;
        then Q^cc ⊆ (Cl(P^c))^c by TOPS_1:15;
        then Q ⊆ (Cl(P^c))^c by TOPS_1:10;
        then Q ⊆ Int P by TOPS_1:42;
        hence thesis by Z3, BOOLE:5;
    end;
```

（ファイル　― 例 No.11 を見てください。　：欠落）

この証明の最初の２つのステップは以下に示す存在仮定（existential assumption）で置き換えることもできます：

given Q **such that** Z1: Q is_open **and** Z2: Q ⊆ P **and** Z3: x ∈ Q;

証明は形（form）をファイル art.lst の例 No.10 にあるように仮定します。他の存在仮定のケースはファイル art.lst の例 Nos. 25, 26, 27, 28 にあります。（: 欠落）

注釈：　存在仮定（existential assumption）で与えられた条件は仮定の形で、つまり語 **assume** *を使って、書くことはできません。　それゆえ以下の記述は誤りです：*

given Q;
　assume that Z1: Q is_open **and** Z2: Q ⊆ P **and** Z3: x ∈ Q;

アネックスの練習問題　― Z13.lst ―　はそのような誤った仮定の帰結を示します（: 欠落）。　同様に選択のステートメント：

consider Q **such that** Z1: Q is_open **and** Z2: Q⊆P **and** Z3: x ∈ Q;

と記述することもできません。　というわけで：

consider Q; **assume that** Z1: Q is_open **and** Z2: Q⊆P **and** Z3:x ∈ Q;

となります。
（annex　― Z12.lst を見て下さい。：欠落）

Ⅳ.3. その他の Mizar 構文（Other Mizar Constructions）

Ⅳ.３.１ 反復等号（ITERATIVE EQUALITY）

さて*反復等号 (iterative equality)*と呼ばれる Mizar 構文を議論しましょう。等号文（equality formulas）の証明で応用例が見られます。　それらの式はある条件を満たさねばなりません、すなわち自由変数を含んでいないということです。　それらの式の変数は(訳注:型が)設定されていなければなりません(must be fixed)。　それは一般化、例示、選択のステートメント、型変更のステートメント、あるいは変数の局所定義により設定されます。

一方、*反復等号*は証明の新しいアイデア（考え）は導入しません。
定理 $(P^c)^c = P$ の証明を検討してみましょう、それは後で*反復等号*の説明に使います。

```
(P^c)^c = P
proof
   (P^c)^c = Ω\ ¥ (P^c) by TOPS_1:5;
   then (P^c)^c = Ω\ ¥ (Ω\ ¥ P) by TOPS_1:5;
   then (P^c)^c = Ω\ ∩ P by BOOLE:82;
   hence (P^c)^c = P by TOPS_1:3;
end;
```

命題とその証明に特徴的な次の事実に注意して下さい。

1. 証明された文は等式（equality formula）です。
2. 証明のすべてのステップに現れる式は等式（equality）です。　さらに等式の左側の項は各ステップで同一です（$(P^c)^c$）。
3. 推論の全てのステップは、２番目から始まりますが、(linking により）先行するステップを参照しています。

これらの事実は命題 $(P^c)^c = P$ の証明が反復等号によって実行される条件を満たします。　それは、このケースでは：

```
(P^c)^c = Ω\ ¥ (P^c) by TOPS_1:5;
     .= Ω\ ¥ (Ω\ ¥ P) by TOPS_1:5;
     .= Ω\ ∩ P by BOOLE:82;
     .= P by TOPS_1:3;
```

の形になります。

この記述の意味するものは？　例えば、推論（inference）

```
     .= Ω\ ¥ (Ω\ ¥ P) by TOPS_1:5;
```

については、表現

```
then (P^c)^c = Ω\ ¥ (Ω\ ¥ P) by TOPS_1:5;
```

が、もうひとつの別の記述になります。　ここで **then** は先行する文、つまり $(P^c)^c = \Omega\setminus$ ¥ (P^c) への参照です。

シンボル .= の前で、先行する推論のステップの justification の後にはセミコロン ; を付けないということを心に留めておかないといけません。　セミコロンは*反復等号*の最後に要求され、それは

```
     .= term justification
```

の形の最後の表現の後です。

上記の*反復等号*は、証明を終結する結論を欠いているので、命題 (P^{c})^{c} = P の完全な証明ではありません。　完全にするためには：

hence (P^{c})^{c} = P

を付け加えれば十分です。　語 **hence** は文 (P^{c})^{c} = P は*反復等号*の形の中で完全な推論（entire reasoning）の参照により justify されていることを意味します。　もちろん結論は **thus** を使う方法で書くこともできます、しかし*反復等号*を参照するために、ラベルで開始（open）する等式を前に置かない（prefix）といけません。　反復等号の前に **thus** を置くこともあります。　さらに結論 (P^{c})^{c} = P は式 **thesis** で置き換えられることもあります。　従って命題 (P^{c})^{c} = P の証明は以下の形をとります。

proof

（語 **proof** *を書いた後に変数* G *と* P *が設定されます。　証明している文はシステムにより* **for** G, P **holds** (P^{c})^{c} = P *という形の量化式として読み込まれるので、語* **proof** *を書いた後でシステムは自動的に一般化* **let** G, P; *を実行し、変数* G *と* P *を設定します。　*P *は型* Subset of G *を持つので、変数* G *も設定されます。）*

```
(P^c)^c = Ω𝔊 ¥ (P^c) by TOPS_1:5;
     .= Ω𝔊 ¥ (Ω𝔊 ¥ P) by TOPS_1:5;
     .= Ω𝔊 ∩ P by BOOLE:82;
     .= P by TOPS_1:3;
    hence thesis;
  end;
```

（ファイル art.lst の例 No.33 を見て下さい。：欠落）

ここに、今考えている命題の証明の他のバージョン：

```
proof
  A: (P^c)^c = Ω𝔊 ¥ (P^c) by TOPS_1:5;
           .= Ω𝔊 ¥ (Ω𝔊 ¥ P) by TOPS_1:5;
           .= Ω𝔊 ∩ P by BOOLE:82;
           .= P by TOPS_1:3;
         thus thesis by A;
end;
```

を示します。　上の証明は最後の２つの文：

```
           .= P by TOPS_1:3;
```

thus thesis by A;

が次の形の単一のステートメント：

hence $(P^c)^c = P$ **by** TOPS_1:3;

あるいは

hence thesis by TOPS_1:3;

で置き換えると、すこし短縮することができます。 証明はファイル art.lst の例 No.34 のような形になります。 (：欠落)

反復等号における推論（reasoning）の種々のステップは（ **by** による）straightforward justification を持ちます。 反復等号では、proof ではない、スキームによる justification もあります。 最も単純なのは、推論のステップが CHECKER にとって自明である時で、そのステップの justification は空（empty）になると思います。

文 $(P^c)^c = P$ は、変数 $\mathfrak{G}$ と P を設定することで選択のステートメントによって反復等号を使って証明することができます。 その場合、語 **proof** と **end** は書いてはいけません。 それは

consider $\mathfrak{G}$, P;
$(P^c)^c = \Omega\mathfrak{G} \setminus (P^c)$ **by** TOPS_1:5;
.= $\Omega\mathfrak{G} \setminus (\Omega\mathfrak{G} \setminus P)$ **by** TOPS_1:5;
.= $\Omega\mathfrak{G} \cap P$ **by** BOOLE:82;
.= P **by** TOPS_1:3;

となるでしょう。

この方法で設定された変数 $\mathfrak{G}$ と P はここで設定された変数に対してのみ証明された文の使用を可能にし、それは P と $\mathfrak{G}$ に対してです（例 No.35 を見て下さい：欠落)。 というわけで、そのような証明の方法は実用的ではありません。 変数が設定されていない場合にはエラーが annex の Z11.lst にあるようにレポートされます（：欠落)。 No.62 のエラーは反復等号のなかでは自由変数が許されないこと、エラー No.140 は未知の変数が存在することを述べています。(：欠落)

*反復等号*は以下のように説明することもできます。

$t_1, t_2, \ldots, t_n, t_{n+1}$ が対応する項で推論（reasoning）が $t_1 = t_2$ & $t_2 = t_3$ & … & $t_n = t_{n+1}$ の straightforward justification であれば、$t_1 = t_{n+1}$; はもう一つの推論、すなわち：

$t_1 = t_2$ *straightforward-justification*

.= t_3 *straightforward-justification*

.....

.= t_n *straightforward-justification*

.= t_{n+1} *straightforward-justification;*

の形の反復等号で等価的に（equivalently）置き換えることもできます。

定理　　Int(Int P) = Int P

は以下のように証明することもできます。

proof
Int P = (Cl (P^c))^c **by** TOPS_1:42;
then Int(Int P) = (Cl (((Cl (P^c))^c)^c))^c &
(Cl (((Cl (P^c))^c)^c))^c = (Cl (Cl (P^c)))^c &
(Cl (Cl (P^c)))^c = (Cl (P^c))^c **by** TOPS_1:10, TOPS_1:42,
TOPS_1:26;
hence Int(Int P) = Int P by TOPS_1:42;
end;

（ファイル art.lst の例 No.31 を見て下さい。: 欠落）

あるいは*反復等号*への参照により：

proof
Int P = (Cl (P^c))^c **by** TOPS_1:42;
then Int(Int P) = (Cl (((Cl (P^c))^c)^c))^c by TOPS_1:42
.= (Cl (Cl (P^c)))^c by TOPS_1:10
.= (Cl (P^c))^c **by** TOPS_1:26;
hence thesis by TOPS_1:42;
end;

次のファイル art.lst の例には*反復等号*の使用による文 Fr P = Fr(P^c) の証明があります。(: 欠落)

Ⅳ.３.２. DIFFUSE STATEMENT

ある命題の証明で、補助的な（複数の）文を justify したほうが便利なことがあります。 その文が by (**by** により)あるいはスキーム（**from** により）により straightforwardly に justify できないときは、証明（すなわち語 **proof** で始まり **end** で終結（concluded）するある種の推論）を実行せねばなりませ

ん。　すると、これは入れ子になった証明になります。　これで証明は明晰さを失うかもしれません。　しかしながら、上に書いた状況に適用できる Mizar 構文があります。　それは

now *reasoning* **end;**

の構文のことです。　推論の多様なステップは証明（proof）のステップと同じ原理の上に構成されます。 proof のなかのこの新しい構文の応用を例で説明しましょう。　命題：

$\mathcal{F}$ is_closed **implies** meet $\mathcal{F}$ is_closed

を考えてみましょう。

これは α **implies** β の形の命題で、

α は 式　$\mathcal{F}$ is_closed,

β は 式　meet $\mathcal{F}$ is_closed

です。　さらに

γ を 式　$\mathcal{F} \neq \emptyset$ としましょう。

この命題は **by** やスキームで証明しようと思ってもできません（be：仮定法)。　そこで proof（証明）を実行しなければなりません。

（γ **implies** β）& （**not** γ **implies** β）**implies** β

の形の式は恒真命題（tautology）なので、proof の中で

γ **implies** β　と　**not** γ **implies** β

の形の２つの補助的な文を証明するのが便利です。　証明（proof）の中でこれらの文はそれぞれ T と K1 というラベルを付けます。　（訳注：ラベル名の混乱 T1 ？）

ここに文の証明を載せます：

```
(●) F is_closed implies meet F is_closed
    proof
        assume F is_closed;
        then A: COMPLEMENT(F) is_open by TOPS_2:16;
        T: F ≠ ∅ implies meet F is_closed
            proof
                assume F ≠ ∅;
                then union COMPLEMENT(F) is_open by TOPS_2:26;
```

```
            hence meet 𝓕 is_closed by TOPS_1:29, A;
          end;
        𝓕 = ∅ implies meet 𝓕 is_closed
          proof
            assume 𝓕 = ∅;
            then meet 𝓕 = ∅(G) by PRE_TOPC:11;
            hence meet 𝓕 is_closed by TOPS_1:22;
          end;
      hence thesis by T;
  end;
```

この証明で式

```
meet 𝓕 is_closed
```

は語 **thesis** で表わされています。
（ファイル art.lst の例 No.37 を見て下さい。　：欠落）

注釈：

　　β　と　（γ **implies** β）＆（**not** γ **implies** β）

という形の式は同一の semantic correlate を持ちません。　こうして

　　β

という形の文の証明のスケルトンは文

　　（γ **implies** β）＆（**not** γ **implies** β）

のケースのように論理積（conjunction）の証明のスケルトンのもとに含める（subsumed）ことはできません。

今、同じ命題を類似の方法で証明しましょう（その証明も文 γ **implies** β と **not** γ **implies** β の justification からなります)、しかし推論の記述は違うものになるでしょう。
命題：

```
    𝓕 is_closed implies meet 𝓕 is_closed
      proof
        assume 𝓕 is_closed;
        then A: COMPLEMENT(𝓕) is_open by TOPS_2:16;
```

があります。　γ ***implies*** β を証明しましょう。　しかしその文はオープンには書かれていません。　それの *justification* という結果になる推論を実行しま

しょう。 Mizar ではそういう構文は語 ***now*** *で始まり* ***end;*** *という表現で終了します。*

```
T1: now
```

(語 **now** *にはセミコロンも* **proof** *という語も続きません。 証明中の文は含意なのでその前項を仮定します。)*

```
    assume 𝓕 ≠ Ø;
```

(下にその命題の justification を誘導する推論のさらなるステップがあります。 それはこの時点で式 β です)

```
    then union COMPLEMENT(𝓕) is_open by TOPS_2:26;
    hence meet 𝓕 is_closed by TOPS_1:29, A;
end;
```

上の推論で文 γ **implies** β の証明を完了します。

その参照を可能にするために、すでに行われているように、 **now** の前にラベルを置きます。 **now ... end;** の推論の後には linking が許されます。

命題のさらに進んだ証明は以下のようになるでしょう：

```
now assume 𝓕 = Ø;
      then meet 𝓕 = Ø by SETFAM_1:2;
      then meet 𝓕 = Ø(Ġ) by PRE_TOPC:11;
      hence meet 𝓕 is_closed by TOPS_1:22;
end;
hence meet 𝓕 is_closed by T1;
```

end;

(その結論の justification で、前に証明された文 γ **implies** *β のラベル* T1 *がそこに現れます。)*

上の証明は annex の例 No.38 に示します。(: 欠落)

語 **now** の前なので、その構文で証明されるべき命題は書きません。

now *reasoning* **end;**

の形の構文は *diffuse statement* と呼ばれてきました（has been called)。

注釈：

1. diffuse statement で証明される命題はその推論のスケルトンを解析することで読み解かれます。 構文 **now ...** *におけるスケルトン作成の原理は証明(proofs)の場合と同じです。 語* **now** *と* **end** *は、ある意味で、それぞれ*

語 **proof** *と* **end** *を置き換えます。*

2．*diffuse statement の式* **thesis** *は即時外部証明 (immediate external proof) の命題を意味します。　式* **thesis** *の使用は、その種の推論がある種の証明に含まれるという条件で diffuse statement を通じて許されます (***thesis** *は proof の中でのみ使用されると思います)。*

3．**now** *の前にラベルを付けると、そのラベルへの参照はそのラベルに続く語* **now** *で始まる diffuse reasoning で証明された命題への参照を意味します。*

4．*diffuse statement の後では* linking *が許されます。*

5．**now** *の前に* **then, thus, hence** *を書くことは許されません。*

6．**now** *で始まるすべての推論 (reasoning) は* **end** *で終わらねばなりません。*

例

α **or** β **implies** γ

の形をした文が証明される文です。　diffuse statement の応用例を持つその文の証明がどういうものかを示しましょう。　式

（α **implies** γ）＆（β **implies** γ）**implies**（α **or** β **implies** γ）

は恒真命題なので、証明の中で補助的な文

α **implies** γ　と　β **implies** γ

を使用する価値があります。　そうすると証明は以下のようになります：

```
proof
    ......
    assume P: α or β;
(今γが命題です。)
    ......
    A: now
        ......
        assume α;
        ......
        thus γ;
        ......
    end;
```

```
        ......
        now
            ......
            assume β;
            ......
            thus γ;
            ......
        end;
        hence γ by A,P;
        ......
    end;
```

いろいろな推論の結論を **thesis** で置き換えられます。 diffuse statement では **thesis** は命題が証明の終わりでそうであるように、 γ を意味します。

例

α iff β

の形の文を証明する時、式

(**not** α **implies not** β) **implies** ((**not** β **implies not** α) **implies** (α **iff** β))

が恒真命題であるという事実を利用すると便利です。 diffuse statement で２つの

not α **implies not** β と
not β **implies not** α

という形の補助的な文を証明することができます。
例として定理 CONNSP_1:11 の証明：

```
G is_connected iff for A,B being Subset of G st ΩG = A ∪ B & A ≠ ØG
& B ≠ ØG & A is_closed & B is_closed holds A ∩ B ≠ ØG
```

を使いましょう。

その場合

```
α    は式    G is_connected
β    は式    for A,B being Subset t of G st ΩG = A ∪ B
                & A ≠ ØG & B ≠ ØG & A is_closed & B is_closed
```

holds A ∩ B ≠ ∅𝔊

not *α* *は式* **not** 𝔊 is_connected
not *β* *は式* **ex** A,B being Subset **of** 𝔊 **st**
Ω𝔊 = A ∪ B & A ≠ ∅𝔊 & B ≠ ∅𝔊 &
A is_closed & B is_closed & A ∩ B = ∅𝔊

となります。 最初の diffuse statement で文

not *β* **implies not** *α*

を、２番目に文

not *α* **implies not** *β*

を証明しましょう。 ここにその証明の形：

```
proof
    T: now given A,B being Subset of 𝔊 such that
        Z1: Ω𝔊 = A ∪ B and
        Z2: A ≠ ∅𝔊 & B ≠ ∅𝔊 and
        Z3: A is_closed & B is_closed & A ∩ B = ∅𝔊;
        .....
        thus not 𝔊 is_connected by ... ;
    end;
    now assume not 𝔊 is_connected;
    .....
    thus ex A,B being Subset of 𝔊 st Ω𝔊 = A ∪ B & A ≠ ∅𝔊 &
      B ≠ ∅𝔊 & A is_closed & B is_closed & A ∩ B = ∅𝔊 by ...;
    end;
    hence thesis by T;
end;
```

があります。 完全な証明はファイル art.lst の例 No.25 にあります（: 欠落)。
他の diffuse statement の例はファイル art.lst の No.26, 27, 28, 29 で見つかるはずです（: 欠落)。

Ⅳ.３.３. 型変更のステートメント (STATEMENT OF A CHANGE OF TYPE)

証明では、時々考えているオブジェクト型の変更が必要になります。 これは、とりわけ(//??amount other things->among other things)ある種の定理は、決まった型のオブジェクトに対してのみ証明されるという事実のせいです。

例えば位相空間に関連するアーティクルでは、その空間の点についての定理があります。　例として文

```
P ⊆ Cl P
```

の証明を書いてみましょう。　それは定義の拡張（definitional expansion）による証明になります。

```
reserve G for TopSpace, x for Any, P, Q, B for Subset of G;
P ⊆ Cl P
    proof
        let x; assume x;  x ∈ P;
```

（今、式　x ∈ Cl P が命題です。）

文

```
(●)  for B being Subset of G st B is_closed holds
      A ⊆ B implies p ∈ B
```

を証明してみましょう（訳注：？がないので Should we prove… は疑問文ではなく倒置による強勢？）。　定理 PRE_TOPC:45 から P ∈ Cl P を得ることができます。しかしその定理が適用されるべきであれば、型 Any を持つ x の indicator は位相空間 G の点、それはその型が Point of G であるオブジェクトです、として取り扱われるべきです。

そのため

reconsider *list-of-changes-of-type* **as** *type* **of** *justification*;

という形の Mizar 構文を利用しましょう。　証明でこの構文を利用して：

```
reconsider t=x as Point of G by TOPS_1:1, x;
```

と書くことができ、それは：

x *を位相空間* G *のある点と考えましょう*

ということを意味します（ここで t は任意の識別子です）。

注釈：　同等化（equalization） t = x *で型の変更をするオブジェクトの識別子は等式（equality）の右側でないといけません。　左側は任意の識別子で、変数の予約で予約されたものを使う必要はありません。*

*型変更のステートメント*は　－**reconsider** … という構文が呼ばれますが－推論の現在のレベルの他の部分（further part）（定数が導入されたレベルです）

において、変数予約で x が型 Any に予約されていたとしても、識別子 x の型は（明示的に与えられなければ）Point **of** 𝔊 となるという結果になります。もちろん、型の変更は適切に justify されなければなりません。　われわれのケースでは定理 TOPS_1:1、それは：

```
x ∈ P implies x is Point of 𝔊
```

ですが、仮定、それは x ∈ P です、を参照しなければなりません。

この証明の次のステップは（•）とマークされた文の justification からなります。　それから主証明（main proof）の結論（conclusion）を書くことだけが必要です。　ここに主証明の完成版をのせます：

```
for B being Subset of 𝔊 st B is_closed holds P ⊆ B implies t ∈ B
        proof
            let Q; assume Q is_closed; assume P ⊆ Q;
            hence t ∈ Q by x, BOOLE:11;
        end;
     hence x ∈ Cl P by PRE_TOPC:45;
  end;
```

（ファイル art.lst の例 No.14 を見て下さい。　：欠落）

今考えている証明の型変更のステートメントは、同等化（equalization）ではない形で記述されることもあります。

x の識別子の型を変更しようと思ったら：

```
reconsider x as Point of 𝔊 by TOPS_1:1, x;
```

のように記述されるでしょう。

従って、前のバージョンの証明での識別子 t をすべて識別子 x に変えなければなりません。　証明は以下のような形になるでしょう。

```
    P ⊆ Cl P
  proof
    let x; assume x: x ∈ P;
    reconsider x as Point of 𝔊 by TOPS_1:1, x;
    for B being Subset of 𝔊 st B is_closed holds P ⊆ B implies x ∈ B
        proof
            let Q: assume Q is_closed; assume P ⊆ Q;
            hence x ∈ Q by x, BOOLE:11;
        end;
```

```
    hence  thesis by PRE_TOPC:45;
  end;
```

最後のステートメントで **thesis** は `x ∈ Cl P` で置き換えることはできません。　なぜならその文は一般化からは `x` について何も言っていないからです、しかし **reconsider** から `x` を参照しています（**reconsider** は一般化（generalization）をオーバーライドしています)。

例 No.13 は型変更のステートメントも含みます。(: 欠落)

型変更のステートメントで導入された定数は、一般化、選択のステートメント、もう一つの型変更のステートメント、例示、存在仮定、そして変数のローカル定義などによりオーバーライドされることがあります。

型変更のリストは複数の同等化（equalization）（あるいは項）の形をとることがあります。　その場合は、お互いにコンマで分離されなければなりません。

BIBLIOGRAPHY

[1] Selected papers on Mizar and their abstracts:
-[PRE_TOPC] Beata Padlewski and Agata Darmochwał. Topological Spaces and Continuous Functions. *Formalized Mathematics,* 1(**1**): 223-230, 1990.
-[TOPS_1] Mirosław Wysocki and Agata Darmochwał. Subsets of Topological Spaces.
Formalized Mathematics, 1(**1**): 231-237, 1990.
-[CONNSP_1] Beata Padlewska. Connected Spaces. *Formalized Mathematics,* 1(**1**): 239-244, 1990.

[2] Raisowa, H., Introduction to Modern Mathematics, Amsterdam-London-Warsaw 1973.

[3] Pogorzelski, W.A., Słupecki, J., O dowodzie matematycznym (Mathematical Proofs), Warsaw 1970.

[4] Pogorzelski, W.A., Klasyczny rachunek kwantyfikatorów (The Classical Functional Calculus), Warsaw 1970.

[5] Engelking, R., Sieklucki, K., Wstęp do topologii (Introduction to Topology), Warsaw 1986.

[6] Trybulec, A., Rudnicki, P., A Collection of $\mathrm{T_EX}$ed Mizar Abstracts, University of Alberta, Canada, 1989.
Trybulec, A., Syntaktyka Mizara (Mizar Syntactics), Białystok 1989.

索 引（INDEX）

A

ABSTR.....20, 57
Any14, 28, 29, 30, 31, 32, 33, 38, 39, 41, 53, 74, 75, 76, 77, 87, 91, 102, 103
assume12, 21, 53, 62, 63, 66, 67, 68, 70, 72, 75, 76, 77, 78, 80, 82, 83, 84, 85, 86, 87, 88, 90, 91, 92, 97, 98, 99, 100, 102, 104
Assumption.....63
attribute.....19
auxiliary lemma.....53

B

being21, 31, 32, 37, 38, 39, 40, 41, 42, 43, 44, 45, 47, 48, 53, 55, 56, 57, 58, 62, 74, 75, 76, 78, 79, 87, 88, 89, 90, 101, 102, 103, 104

C

CHECKER.....53, 57, 64, 87, 95
commutativity.....71
conjunction.....18, 49, 64, 65, 67, 69, 71
consider21, 31, 86, 87, 88, 89, 90, 91, 92, 95, 103, 104
contradiction21, 36, 49, 50, 51, 63, 82, 83, 84, 85, 86, 88
cut down.....74

D

definiens.....19, 26
definitional expansion.....18, 76, 102
definitions.....8, 18, 19, 21, 76
DICT.....19, 20
diffuse statement.61, 73, 99, 100, 101, 102
direct justification 52
disjunction 64, 72

E

equality 19, 35, 92, 103
equalization 80
equivalence 49, 53, 64, 71
exemplification 61, 79
existential assumption 80, 90, 91, 92
EXISTENTIAL SENTENCE 79

F

Frænkel’s operators 24

G

general sentence 44, 64
Generalization 63
given 21, 90, 91, 92, 102

H

homonymous 25

I

inclusion 18, 19

J

justification52, 53, 54, 57, 60, 61, 64, 66, 72, 80, 87, 93, 95, 98, 99, 103

L

let15, 16, 21, 26, 29, 30, 31, 32, 53, 56, 58, 62, 73, 74, 75, 76, 77, 78, 79, 88, 94, 102, 104
LIBRARIAN 20

linking.....61, 67

M

MIZAR.....19, 57
mplication.....44, 64, 75

N

negation.....49, 68
now21, 34, 53, 60, 63, 84, 96, 98, 99, 100, 102
NUMERALS.....22

P

PREL.....20
premises.....52, 61

Q

qua.....21, 27, 35
quod erat demonstrandum.....55

R

REASONER.....64
reasoning52, 54, 55, 56, 57, 58, 60, 61, 63, 76, 89, 93, 95, 96, 99, 100
reconsider.....104

S

schemes.....8, 19, 21
semantic correlate45, 49, 50, 51, 63, 65, 68, 69, 72, 79, 82, 98
semantic form.....49, 51, 68, 69
Simple justification.....52
statement of choice.....87
Straightforward justification.....52
synonym.....21, 50

T

TARSKI 8, 12, 18, 19, 59, 76
then21, 61, 62, 73, 88, 89, 90, 91, 92, 93, 96, 97, 98, 99, 100
theorem 8, 17, 19, 20, 21, 55, 59

V

VERUM 49, 82, 83

あ

ASCII 5, 6, 11, 27
後項 36, 66, 75, 83, 84, 85, 86
アブストラクト 20, 25
アポストロフィ 7, 12, 13
アンダースコア 7, 12, 13
暗黙的 37, 38, 39

い

一般化44, 45, 54, 63, 64, 73, 74, 75, 76, 77, 78, 79, 88, 89, 92, 94, 104
一般化文 44, 73, 74

お

オーバーライド13, 16, 17, 60, 76, 80, 89, 104

か

開集合 9, 15, 42, 43, 44
拡張子 7, 8, 19, 20, 57
カッコ10, 11, 14, 23, 24, 25, 27, 29, 30, 31, 36, 45, 46, 51, 68, 69
合併集合 36, 42, 44
仮定12, 18, 52, 54, 56, 61, 62, 63, 66, 67, 68, 71, 72, 73, 75, 76, 77, 78, 79, 80, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 97, 98, 103, 104
空集合 9, 15, 43, 44

含意18, 36, 44, 46, 51, 53, 64, 66, 68, 69, 71, 72, 75, 77, 78, 80, 81, 83, 84, 85, 86, 87, 88, 90, 98
環境の指令 7, 59
環境部 13, 15, 16, 17, 28, 56, 58
間接証明55, 62, 63, 75, 82, 83, 84, 85, 86, 88

き

帰納法 19
規約 5, 13, 32
境界 9, 42, 43, 44
共通集合 15, 16, 17, 36, 43, 44, 46, 59
共通部分 18, 59
切り落とし 74, 75, 80

け

言語プロセッサ 16, 20, 46, 49
原子式 34, 35, 36, 49

こ

交換律 71
合接 36
構造体 10, 26, 27, 29, 41
拘束力 8, 11, 45
コレクティブ・アサンプション 67
コンパクト集合 10

さ

再定義 16, 17
参照8, 12, 13, 18, 20, 25, 52, 54, 57, 58, 60, 67, 72, 74, 87, 93, 96, 99, 103, 104

し

signature 指令 13, 15, 16, 17
自然数 6, 11, 28
集合体 27
集合の族 34
修飾項 27
修飾された変数 38
修飾子（qualifiers） 8
自由変数 95
述語式 34, 49, 50
述語のシンボル 9, 22, 34, 35, 58
順序対 24, 25
シングル・アサンプション 67
シングルトン 24

す

推論52, 54, 56, 57, 58, 59, 60, 61, 63, 76, 80, 89, 90, 93, 95, 96, 98, 99, 100, 101, 103
スキーム7, 8, 10, 15, 19, 20, 54, 87, 95, 96, 97
スケルトン50, 54, 63, 64, 65, 66, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 98, 99
スコープ 39, 41, 56

せ

セグメント 37, 38
接続子 36, 67
絶対値 23
ZF 集合論 34
セレクター項 26
前項36, 46, 66, 69, 75, 77, 78, 80, 83, 84, 86, 87, 90, 91
全称文 39, 48, 49
全称量化子 49
選択子 10
選択のステートメント61, 76, 80, 87, 89, 90, 91, 92, 95, 104
前提 18, 57, 61

そ

存在仮定.....80, 90, 91
存在修飾子.....43
存在文.....79, 80
存在量化子.....80

た

台集合.....10, 27
退縮.....49, 68

ち

抽象化クラス.....49
直積.....14, 23
直接証明.....63, 64, 66, 75, 82, 86

て

定義5, 7, 8, 9, 10, 14, 15, 16, 17, 18, 19, 20, 25, 26, 27, 28, 29, 30, 31, 49, 58, 59, 76, 80, 89, 92, 102, 104
定義の拡張.....19
データベース.....8, 15, 19, 20

と

同意語.....50
同音異義語.....25
等価式.....35
統語構造.....30
統語接続子.....34, 36
統語的関連.....82
同値19, 36, 45, 49, 51, 53, 64, 68, 71, 72, 78
同値関係.....49
同等化.....80, 81, 103, 104, 105

に

２項演算.....15, 16

は

反復等号 92, 93, 95, 96

ひ

左側引数 15, 34
Hidden ボキャブラリ 14
被覆 10
標準の優先度 11

ふ

ファイル名 8, 13, 19
functor カッコ 24
functor シンボル.....8, 22
部分集合6, 9, 14, 16, 17, 18, 33, 40, 41, 42, 43, 44, 55, 56, 59
プライベート functor.....12
フレンケル・オペレータ 24, 25

へ

ペア 23, 24, 50, 51, 60, 72
閉演算 18
閉集合 9, 43, 44
閉包 9, 11, 18, 40, 44, 55, 59
変数12, 13, 16, 22, 24, 25, 27, 28, 31, 32, 33, 37, 38, 39, 40, 47, 48, 49, 50, 56, 73, 74, 75, 76, 80, 89, 92, 94, 95, 103, 104

ほ

ボキャブラリ指令 8, 58
補集合 11, 43, 88

み

右側引数 15, 34, 44
Mizar の予約語.....7, 12, 20

む

矛盾 36, 83, 84, 85

め

明示的 37, 38, 39, 103
命題結合 36

も

モード 10, 15, 16, 19, 26, 28, 29, 30, 31, 41, 42, 58, 59

ゆ

有限数列 10, 23
優先度 11, 15

よ

予約シンボル 12, 20

ら

ライブラリ 12, 13, 20, 52
ラベル 12, 13, 52, 57, 60, 61, 67, 72, 74, 90, 94, 97, 99

り

離接 36
量化子 36, 39, 41, 43, 44, 45, 46, 48, 49, 74, 75, 76, 80
量化式 32, 34, 37, 39, 40, 46, 75, 94

れ

例示 54, 61, 63, 64, 76, 79, 80, 81, 82, 89, 92, 104
レファランス・リスト 53, 54
連結 42, 44
連続写像 9

ろ

ローカル参照 13
ローカル定義 26

訳者あとがき： 数学入門を目指して信州大学 IT 大学院へ入学して Mizar に触れてから早一年。　指導教官のしろだぬき先生、先輩の Zton 氏の教えを受けながら勉強をすすめ、現代数学という思想の輪郭がおぼろげながら見えてきた気がします。　本テキストは位相空間論を例題として扱っており、数学という思想を語る言葉としての集合論を勉強中の小生には興味深いものでした。
(//??...->...) は訳者が明らかに typographical error であると考えるものを修正したもので、//?? に続けて原文の表現を記入し -> の後に変更後の表現を置きました。　訳者の Mizar の知識は不十分ですので、読者の責任で使用ください。Mizar 学習のための個人的使用だけを想定して公開します。

2008 年春　l'Hospitalier