Mizar Users Group

An Outline of PC Mizar

Michał Muzalewski

Series Editor Roman Matuszewski

Fondation

Philippe le Hodey

Brussels, 1993

日本語版（5.1校）by l'Hospitalier
2008

すべての練習問題は PC Mizar system でチェックされています。
ポーランド語から英語へ Olgierd A. Wojtasiewicz によって翻訳されました。

（英語から日本語へは l'Hospitalier によって2008年早春に翻訳されました）

Orders should be addressed to:

Foundation of Logic, Mathematics and Informatics
Mizar Users Group
Krochmalna 3 m. 917
00-864 Warsaw
Poland

Fax: +48(22)624.03.49
e-mail: romat@mizar.org

警　告

Mizar 仕様変更に伴い本書の内容は翻訳時（2008/1）の PC Mizar システムのバージョン 7.8.08、MML 4.95.999 に適用できない部分が多数あります。　読者各位は Mizar Home Page http://mizar.org/ などを参照して最新の情報を入手してください。

目　　次

序

第１部：言語

定　義　(Definitions)

データベース (Data Base)

第２部：　システム

序　文

われわれの目標(object)は Mizar article の書き方を学ぶことです。 “Mizar” という言葉にはあまり明瞭な定義がありません。

1．Mizar は数学の形式化のための言語です。
2．Mizar はアーティクルの処理に関連するコンピュータシステムです。

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 言語の原著者です。 *An Outline of PC Mizar* では言語とシステムの順に議論します。 A. Trybulec と Cz. Byliński には数回にわたり素稿を読んでいただき、たくさんの誤りを指摘していただきました。 いつも激励されました。 とても感謝しています。 このテキストを書くにあたり A. Trybulec の Mizar 関係の多くの著作、そしてこの言語とシステムに捧げられた彼の講義に依拠しました。

また、タイプ原稿を読み、テキストが Mizar アプリケーションの初心者にわかりやすいようにするための貴重な意見をいただき、一緒に議論した Krzysztof Prażmowski に感謝します。

第一部：言語

1．アーティクル（Article）

Mizar article においては標準的な数学論文と同様に、新しい概念を導入し、それらの定義の正当性を証明し、定理を証明します。 これらの定理の証明は、以前に他のアーティクルで証明された定理を参照して、このアーティクルで使用するか、あるいは直接 **TARSKI** と **AXIOMS**（それらは言語的正当性という立場から単独でシステム Mizar によってチェックされています）に基づいて形式化された公理に基づいています。

すべてのアーティクルは：

environ *environment* **begin** *text* **begin** *text* **...**

という形式（form）を持ちます。

environment は Mizar の実在のライブラリから、必要な情報を引き出すための一連の指令手順です。 そういった情報の基礎に立ってテキストで導入された定義の正当性を justify し、定理の証明を行うことができます。 指令（directives）については「指令」の章でより詳細な記述を見いだすことができます。

さらに、ある種の語は太字で書くことにしましょう。 それらは Mizar が予約しているもので、それらの意味は Mizar 言語が定義として厳密に規定しています。 タイプフェースの違いによる表記上の区別は読者の注意を引き、最終的にはそれらを記憶しやすくするためのものです。

テキストは定義、定理、その証明を含みます。 もう少し厳密には、テキストの単位は予約、定義、再定義、スキーム、そして(//??end->and)定理などで、Mizar 文法ではステートメントと呼ばれる術語が使われます。

今ここで、Mizar 文法の観点から、証明 ― もっと広く言うと ― justification、は定義、再定義、スキーム、そしてステートメントの内在的単位（immanent units）であると述べておくことは意義があると思います。 別の言葉で言うと、今述べた文法上の単位は、その justification と一緒になってある数学的内容を表します。

例として *definition* という語の意味を考えて見ましょう。 *definition* の内容は *definiendum*（定義される表現、 expression to be defined）と *definiens*（定義をする文、the defining expression）から成り立ちます。 それゆえ、*definition*: *definiendum* *definiens* *proof of correctness* です。
同様に、*statement* はあるもの（a certain）とその justification の両方を意味

します。

*アブストラクト*はアーティクルのもう一つの形です。　これは Mizar article からすべての証明、全ての補題、全てのプライベート・オブジェクトの定義を除去したものとして得られます。　パブリックの定義と定理だけが残されます。他のアーティクルを利用するときに見るのは、このアブストラクトだけです。Mizar article から*アブストラクト*に変換されたものだけが、これに後続するアーティクルで使用されることになるでしょう。

基礎的言語学的構成

2．型（Types）

Mizar ではあらゆる変数に型があります。 Main Mizar Library の内容について知識を得るには、どの型が使用可能かを知ることが要求されます。
新しい型は構文の定義とモードの定義により導入されます。

変数の型は予約により（訳注：アーティクル）全体に（globally）設定（fixed）されます、例えば

```
reserve X for set, D for DOMAIN;
```

で予約します。

型が予約されていない変数を用いて文を定式化（formulate）する場合には、型を明示的に記述する必要があります、例えば：

```
for X st X <> ∅ ex a being Any st a ∈ X;
```

において **X** は暗黙のうちに限定（修飾）された（qualified implicitly）変数です： その型は以前に予約されています； **a** は明示的に限定（修飾）された（qualified explicitly）変数で、文の中で与えられた **Any** 型を持ちます。

型（types）は*引数*（*arguments*）を持つことがあります。 すなわち、型は他のオブジェクトから導出（derived）されます。 型のシンボルとその引数は **of** という語で分離されます。 例えば：

```
Subset of X,  Element of D,  Function of X, Y
```

などと書きます。
さて、ここで手短に Mizar の基本型について議論しましょう。 Mizar の基本型は **HIDDEN** アーティクルで見ることができ、この **HIDDEN** はすべてのアーティクルに自動的に結合されます。 これらの自動結合される型はビルトイン型と呼ばれます。 **HIDDEN** というアーティクルには：

```
Any;  set;  Element of X;  DOMAIN;
Subset of D;  SUBDOMAIN of D;  Real;  Nat;
```

という型が記述されています。 ここで **X** は **set** 型をもち、**D** は **DOMAIN** 型を持ちます。

Any という型は Mizar で最も広範囲の型です； 他のすべての型は **Any** 型まで拡大されます。

set 型は **Any** 型と同値です。
DOMAIN 型は空でない集合（**non-empty sets**）を意味し、いわゆる定義域（domains）です； **SUBDOMAIN of D**（定義域の部分集合）、**Real**（実数）、**Nat**（自然数）などです。 別のアーティクル α で導入された θ 型を使うためには環境部に：

vocabulary α **; signature** α **;**

の２つの指令が前もって記述されている必要があります
上の記述は、**HIDDEN** アーティクルですでに導入されているビルトイン型には適用されません。

３．項（Terms）

Mizar 言語では、標準的数学におけるのと同様、関数が定義されます。 より正確には、定義されるのは関数の名前で、それは functors（関数記号名）とよばれます。 functors のシンボルは項の構築に使われます。 数学では、項はいくつかの functor のシンボルを繰り返し使用して表現されます：変数と定数は項です。 さらに $\tau_1, \ldots, \tau_p$ が項であれば、そして φ が p 引数の functor であれば $\varphi(\tau_1, \ldots, \tau_p)$ もまた項です。
例えば **x, y** が変数で、そして **+** が２項の functor であれば：

x, y は 0 次（degree 0）の項で、
x + y, y + x は１次の項で、
x + (x + y), (x + x) + (x + x) は２次の項...といった具合です。

Mizar 言語では項の概念はより広範囲に解釈されます。 例えば Mizar 文法は **it** という特別な項を認めます。 項 **it** はモードや functor の定義文（definiens）の中で定義されるべきオブジェクト、つまりモードや functor そのものを示します。 （項 **it** は定義文の外で使ってはいけません）。 **it** を使用して記述される文は簡単で、短く、読みやすいものです。 もっとも重要な５種類の項：

- 変数の識別子
- パブリックの functor を持つ項
- プライベートの functor を持つ項
- フレンケル・オペレータを持つ項
- 集合体（aggregates）

の指摘に限って話を進めましょう。
パブリックの functor を持つ項の引数は：

- functor シンボルの左側、
- functor シンボルの右側、
- 同時に functor シンボルの両側。

に書かれると思います。 もし functor シンボルの片側の引数が２つ以上なら、引数はコンマで区切らねばなりません。 そして引数のリストは：

φ , x φ , φ x , x φ y , (x_{i1}**, … ,** x_{ik}**) φ (** x_{j1}**, … ,** x_{jl}**)**

と丸カッコの内側に書かねばなりません。 ここで φ は functor シンボルです。
プライベートの functor を持つ項の引数は右側にしか書けませんし、引数の数が０の場合でも丸カッコは：

F() , F(X) , F(x_1**, … ,**x_n**)**

と必須です。 ここで **F** はプライベート functor の識別子です。
パブリック functor を持つ項に戻りましょう。 既に述べたように、そういう項の引数は functor シンボルの片側に書かれるか（左側はプリフィックス記述で右側はポストフィックス記述）、あるいは同時に両側に書かれます（インフィックス記述）。 ですから

bool X, ∪X, -1, graph f, dom f, rng f, id X, ~f, Funcs(X,Y), χ(A,X), len p, Plane(A,B,C)

はプリフィックス・ターム（前置項）で

x″, x*, x!

はポストフィックス・ターム（後置項）で、

x+y, f|X, f″X, f.(a,b)

はインフィックス・タームとなります。 もう一回言いましょう(//??sad -> said)：

- 単一の引数は、必須ではありませんが、カッコが必要なことがあります。
- ２つあるいはそれ以上の引数にはカッコが必要です。

例えば：**(x)″**と書くならば **x″**で十分です。 他方 **Funcs(X,Y)** のかわりに **Funcs X,Y** は許されません。

上で議論したパブリックの functor をもつ項は functor シンボルと引数でできています。 Mizar では、いわゆる functor bracket という、functor の次にくるものと、その引数を書くことができる項が許されます。 シンボル [と] 、{ と } 、そして [: と :] は左と右の functor bracket の例です。 ここに数学的オブジェクトとそれらのオブジェクトを書いた項を挙げます。 これらは上記のカッコを使ってつくられました：

１．順序対、３対子、４対子、さらには：
[x1,x2], … , [x1,x2,x3,x4,x5,x6,x7,x8,x9] を含む９対子（nonatuples）までの n 対子。

２．Singleton（集合で単数のもの）、対、そして：
{x1}, {x1, x2}, {x1,x2,x3}, … , {x1,x2,x3,x4,x5,x6,x7,x8} を含む８要素の集合までの有限集合。

３．集合の直積（Cartesian product）：
[: X1,X2 :], [: X1,X2,X3 :], [: X1,X2,X3,X4 :].

最も単純なフレンケル項は、{ *term* : *sentence* } の形式の表現です。 ここにフレンケル項の例を記します。

```
{p~ : p is Element of [:the point of M, the point of M:]}
```

この場合変数 **p** は points の順序対を、 **p~** は要素 **p** の同値類（equivalence class）を表します。 ですから上記の項は構造体 **M** のすべての自由ベクトルの集合を意味します。 最後に functor を特定する原理の定式化をしてみましょう。 それは functor がどのように識別されるかという問いに対する回答です。システム Mizar は functor を

- シンボル、
- 引数の数、
- 引数の型、

で識別します。 ２つの functor のシンボルが異なるか、引数の数、あるいは引数が一つのときはその型が、お互いに異なればシステムはそれらの functor を別のものとして取り扱います。

４．原子式（Atomic formulas）

Mizar 言語では、標準的な数学と同様、関係が定義されます。　より正確には定義されるのは関係の名前、すなわち述語です。　述語は原子式を構築するのに使われます。　そして次にそれは　　― 連結詞（connectives）と量化子（quantifiers）を使用して ―　　任意の式（arbitrary formulas）を構築するために使用されます。

さしあたり原子式の議論をしましょう。　原子式の基本となる種類は：

- パブリック述語を持つ原子式、
- プライベート述語を持つ原子式、

です。　前者では引数は：

- 述語シンボルの左側
- 述語シンボルの右側、あるいは
- 同時に述語シンボルの両側

に書かれるでしょう。
引数のリストはカッコで囲まれることなく：

Π, x Π, Π x, x Π y, $x_{i1}, \ldots, x_{ik}\Pi x_{j1}, \ldots, x_{jl}$

となります。　ここで Π は述語のシンボルを表します。　特に、パブリック述語を持つ原子式は：

x = y　そして　**x <> y**

と等式型の式（equational formulas）を含みます。
後の場合（プライベート述語を持つ原子式の場合）では、引数は右側に単独で書かれます。　そして（角）カッコは引数の数が 0 の場合でも：

P[ ], P[x], **P[**$x_1, \ldots, x_n$**]**

のように必須です。　ここで **P** はプライベート述語を表します。
さらに、原子式は限定修飾された式（qualifying formulas）を含み、それは *term* **is** *type* の形の表現となります。　例えば：

√(b² − 4ac) is real
GroupsStr ≪REAL, addreal, compreal, 0≫ is AbGroup

などです。

functor の場合のように、システムは与えられた述語をシンボル、引数の数、そして引数の型で識別(//??identifiers->identifies)します。　システムは、シンボル、引数の数、引数が一つの場合はその型の識別などの手段により、お互いに異なるどんな２つの述語も異なる述語として扱います。

５．式　(Formulas)

否定、論理積、論理和、含意、同値はそれぞれ **not, &, or, implies, iff** で表されます。

上の順番はそれらの連結詞（connectives）の結合（binding）の強さが降順に対応します（訳注： **not** が最強）。　但し、含意と同値は同じ結合の強さです。論理積と論理和については連結した記述が許されます。　つまりシステム自身が失われたカッコを付け加えます。　例えば式

ϕ_1 **&** ϕ_2 **&** ϕ_3

はシステムにより

(ϕ_1 **&** ϕ_2**)** **&** ϕ_3

に変換されます。　これと反対に **implies** と **iff** の場合は連結した記述は許されません。　例えば

ϕ_1 **iff** ϕ_2 **iff** ϕ_3

はエラーとなります。

３種類の量化文（quantified sentences）：

１．単純な全称（命題）式、例えば

```
for x being Any holds x = x;
```

２．制限された全称（命題）式（universal limited formula）、例えば

```
for x being Real st <> 0 holds x•x″ = 1;
```

３．存在（命題）式（an existential formula）、例えば

```
ex x, y st x <> y;
```

があります。　量化式に先行する **hold** という語は省略されることもあります。ですから、例えば

```
for x,y being Nat st x<y holds ex z being Real st x<z & z<y;
```

は

```
for x,y being Nat st x<y ex z being Real st x<z & z<y;
```

で置き換えられることもあります。 ユーザは、Mizar では量化子の結合は連結詞が行う結合より弱いという警告を受けます。 かくして

```
for x holds α & β
```

は

```
for x holds (α & β)
```

と同じことを意味し、

```
(for x holds α) & β
```

ではないのです。

定理の証明（Proving of Theorems）

６．単純ジャスティフィケーション（Simple justification）

今度はいろいろな種類の単純な justification の議論をしましょう。

１．前に真であると仮定されたか、既に justify された文を参照するときは

by ϱ_1**, … ,** ϱ_n**;**

と書きます。　ここで ϱ_1**, … ,**ϱ_n**;** は参照（訳注：参照される文）です。

例

```
X = Y by A, TARSKI:2, BOOLE:def 1;
```

ここで

● プライベート参照 **A** はこのアーティクルの前の方ですでに証明された文のラベルです。

● ライブラリ参照 **TARSKI:2** はアーティクル **TARSKI** の２番目の定理です。

● ライブラリ参照 **BOOLE:def 1** はアーティクル **BOOLE** の最初の定義定理（definitional theorem）です。

２．スキーム （訳注：原文は schemaですが Mizar には schemaという術語がないので以下すべて //??schema-> schemeとします。）を参照するときは、

from *identifier-of-scheme*(**ϱ_1, … , ϱ_n);**

と書きます。 ここで ϱ_1**, … ,**ϱ_n は参照です。 これは前提を持つときで、スキームが何の前提も持たない時は

from *identifier-of-scheme;*

となります。

例

```
ex Z st x holds x ∈ X iff x ∈ X & x ∈ Y from Separation;
thus thesis from FuncEx(A1, A2);
```

Separationと**FuncEx**という語はスキーム（schemes）の識別子です、一方 **A1, A2** は前提の文のラベルで、それはスキーム **FuncEx** の仮定です。

それらの文はそのスキームによって justify するために、最初に証明されねばなりません。　これとは逆にスキーム **Separation** は、前提を持ちません

３．書き上げた文が **CHECKER**（*14. CHECKER の基礎情報*を見て下さい）にとってわかりきった（obvious）ものであれば、justification は全てセミコロンの形、例えば

```
x c= y & x = a & y = b implies a c= b;
```

となります。　Mizar では、二番目のシンボル **c=** は集合の包含を意味します。

４．与えられた文に先行する文を参照する場合、**then** と書けば十分です。そこで

```
A: X c= Y;
B: Y c= Z;
X c= Z by A, B, BOOLE:29;
```

と書く代わりにもっと簡単に

```
A:  X c= Y;
Y c= Z;
then X c= Z by A, BOOLE:29;
```

と書きます。　表現 **then** は linking を意味する術語です。　linking を使えば２つ目のラベルを不要にできます。

７．“命題”式、証明のスケルトン、推論の結果

(Formula “thesis”, skeleton of the proof, and results of reasoning)

thesis という言葉は*証明されるべく残されている*という意味です。　証明の最初の **thesis** がその証明の命題（thesis）を意味することは自明です。Mizar は入れ子の証明、つまり proof の中での proof を許します。　その証明の命題は時に main thesisと呼ばれます。　ですから main thesis はダイナミックに変化する式 **thesis** の初期値です。　**thesis** の値はテキストが **proof** から **end;** に進むにつれて変化し計算されなおします。

１．*α* **implies** *β* という含意命題を証明するなら、その前半の項を真と仮定

し、

assume α;

とすると、**thesis** は後半の項βを意味します。

２．論理積 **α & β** を証明しようとしている時、結論を

thus α;

と書きます。 **thesis** は文βを意味します。

３．全称（命題）文 **for x being θ holds β** を証明しようとしていて、一般化を

let x be θ;

と書けば **thesis** という言葉は βを意味します。

４．存在（命題）ステートメント：

ex x being θ st β

を証明しているなら、例証（exemplification）

take x;

を書くと **thesis** はβを意味します。

これらの４つの構成は、証明のスケルトンの記述から：

- 仮定（assumption）（１）
- 結論（conclusion）（２）
- 一般化（generalization）（３）、そして
- 例証（exemplification）（４）

です。 それらは、まるで main thesis から初期部分を切り落とすかのように振舞い、それに対応するかのように thesis 式は、その値を変化させます。

1'. 仮定 **assume α;** は文 **α implies β** から表現 **α implies** を切り落として、表現 **β** のみを残します。

2'. 結論 **thus α;** は論理積 **α & β** から表現 **α &** を切り落として、表現βのみを残します。

3'. 一般化 **let x be θ;** は全称（命題）文 **for x being θ holds β** か

ら表現 **for x being** θ **holds** を切り落とし表現 β のみを残します。

4'. 例証 **take x;** は存在（命題）文 **ex x being** θ **st** β から表現 **ex x being** θ **st** を切り落とし表現 β のみを残します。

例

次の Mizar article の断片を解析してみましょう；

```
reserve x,y for Any;
theorem T23: (ex x for y holds α) implies (for y ex x st α)
   ::thesis = (ex x for y holds α) implies (for y ex x st α)
proof
  assume ex x for y holds α;
            ::thesis = for y ex x st α
  then consider y such that E: for y holds α
            ::thesis = for y ex x st α
  let y;
            ::thesis =  ex x st α
  E': α by E
            ::thesis =  ex x st α
  thus thesis by E';
            ::thesis = VERUM    （訳注：ラテン語で‘真正の’）
end;
```

（:: から行末までは Mizar テキストのコメントで PCミザールでは処理されません）

文

then consider x such that E: for y holds α**;**

は仮定された存在（命題）ステートメント

ex x for y holds α **;**

を述べ、選択のステートメント

consider x such that E: for y holds α **;**

を引き出し（yields）ます。　このステートメントはステートメント **E** が保持する変数　**x** の値を選択します。

上に書いた、仮定 **assume ex x for y holds** α**;** の次の推論において、一般化

let y;

と結論

thus thesis by E′;

は証明のスケルトンを構成し **thesis** を変化させますが、選択のステートメント

then consider x such that E: for y holds α **;**

とラベルのついた文

E′: α **by E;**

は **thesis** を変えません（それらはいわゆる推論の補助的記述を構成します）。

証明中に **end;** が出現したときに、変化する **thesis** 式（訳注：の値）は VERUM（正確には： **not contradiction**）であるべきだというのは自明です。（訳注：verum はラテン語で‘真正の’）

そうでないと Mizar は **REASONER #70**（*証明すべき部分が残っています*）のエラーをレポートします。

さて、簡単のために proof を終わりから逆に読んでいくと結論 **thus** β はオープン・フォーム（オープン・フォームとは **thus thesis** を意味します）を持ちます。 **end;** から **proof** に向けて読みながら、文 β の左側に適切な表現：

1”. 仮定 **assume** α**;** 文 β に α **implies** を付け加えて、 α **implies** という表現を構成します。

2”. 結論 **thus** α**;** 文 β に表現 α **&** を付け加えて、 α **&** β を構成します。

3”. 一般化 **let x be** θ**;** 文 β に表現 **for x being** θ を加えることで表現 **for x being** θ **holds** β を構成します。

4”. 例証 **take x;** 文 β に表現 **ex x being** θ **st** を付け加えることで、表現 **ex x being** θ **st** β を構成します。

を書くことで、変化する推論の結果をダイナミックに再構成します。

例

次の証明の書かれた紙片を発見したと想像してみて下さい。

proof

```
  assume α ;
  let x be θ ;
  take y;
  thus β ;
end;
```

残念ながら、その紙には証明が終わった定理そのものは記録されていませんでした。　証明を逆方向に読むことで、簡単に推論の結果を見つけることができます。　それが

```
α implies for x being θ ex y st β ;
```

という文であるのは明白です。

唯一つ、変数 **y** の型の再構成には成功しません。　それは先ほどの紙片に見られた証明のステートメントに先行する予約（reservation）で設定されているからです。　証明のスケルトンを作る方法や **thesis** の式を計算する実際の方法は次に実例を参照しながら詳細に説明しましょう。　これは定理を証明するためのいろいろな戦術の議論に有効です。　他方、推論の結果の計算の問題はもう少し後に、diffuse ステートメントについての議論に関連して再度取り上げましょう。　これはそれらの命題が open でないということに由来します。

例
この章の結論として最も短いステートメントを挙げます。

```
not contradiction proof end;
```

contradiction という言葉は論理定数 ***偽*** を意味します。　**not contradiction** が論理定数 ***真*** を意味するのは明らかです。　Mizar 言語においては **contradiction** と **not contradiction** は文と解釈されます。

８．　定理の証明の戦術（The tactics of proving theorems）

定理証明の基本原理は、はじめに完全な推論を一度に書き上げるのではなく、そのスケルトンだけを書くことが大事です。　すなわち

- 仮定（assumption）、
- 結論（conclusion）、そしてまた
- 全称（命題）文を証明するのなら、一般化（generalization）、そして
- 存在（命題）文を証明するなら例証（exemplifications）

です。　初心者は特に証明のスケルトンを作ることを難しいと考えるかもしれ

ません。　スケルトンを作る全ての方法、というわけには行きませんが、多くの方法を示そうと思います。　ここでは定義の拡大（definitional expansion）と ***per cases*** による証明は無視することにします。　物事を単純にするために（//??natters -> matters、訳注：natterはverb） **by** と **from** による単純 justification も除きました。　その証明の構成法は、すでに認証されたスケルトンへ単純な justification を付け加えるかぎりは、自然で推奨される方法です。さあ書き始めましょう。　最初にこの証明の定理を定式化しなければなりません。　そこで、

proof … end;

と書きます。　もちろんドットは一連の推論（reasoning）ステップで置き換えられます。　最後のステップはしばしば

thus thesis;

となります。　推論のステップはセミコロンで分離されます。　それらのステップが直接証明か、あるいは背理法か、それから証明すべき命題自身に依存するのは自明です。　背理法による **thesis** α の証明は含意の証明

not α **implies contradiction**

に帰着（reduces to）します。　仮定による *Jaśkowski の証明*に慣れている読者は *Jaśkowski システム*や Mizar で使用される法則の収斂（convergence of the rules）に気づくでしょう。　それは特に矛盾による証明（訳注：背理法）の場合に適用されます。

さて続けて次の形の命題を検討しましょう。

1．　命題が論理積、
2．　命題が含意、
3．　命題が同値、
4．　命題が論理和、
5．　命題が全称（命題）文、
6．　命題が存在（命題）文

1から6の場合の証明を書く戦術について一つずつ議論しましょう。

8．1　論理積の証明

証明の中で論理積の個々の構成要素が真であることを述べなければなりません。　こうして命題が次の形

C_1 **&** C_2 **&** C_3

であるとき証明は次のようになるでしょう。

```
proof
  thus C1 …;
  thus C2 …;
  thus C3 …;
end;
```

thus という語はそれに続く文が*結論(conclusion)*であることを示します。 *結論*とは、証明の命題（thesis of the proof）かあるいはその一部であることを意味します。 ドットは単純な justification か、あるいは証明を意味します。

例えば：

```
T1: X = Y
  & (ex Z st for x holds x ∈ Z iff x ∈ X & x ∈ Y)
  & for x holds x ∈ X ∩ Y iff x ∈ X & x ∈ Y
proof
  thus X = Y by A, TARSKI:2;
  thus ex Z st x holds x ∈ Z iff x ∈ X & x ∈ Y from Separation;
  thus thesis proof … end;
end;
```

ここで **thesis** という語は証明中の論理積の３番目の構成要素、すなわち文 **for x holds x ∈ X ∩ Y iff x ∈ X & x ∈ Y** を意味します。

８．２ 含意の証明

戦術１－直接法

推論の最初のステップは、

assume *the-antecedent-of-the-implication（含意の前提）；*

そして最後は

thus *the- consequent-of-the-implication（含意の帰結）；*

です。 語 **assume** は証明の中で行われる仮定を示します。 仮定は単一の文であったり（いわゆるシングル・アサンプション）あるいは、ラベルがついて、**and** でひとまとめに結合された一連の複数文（いわゆるコレクティブ・アサン

プション、複文の仮定）であったりします。　後者の場合、仮定の分割は、集合的な仮定（訳注：コレクティブ・アサンプション）の部分的な参照を可能にするので、便利なこともあるのに注意してください。

```
x ∈ X & x ∈ Y implies X ∩ Y <> ∅
  proof
    assume x ∈ X & x ∈ Y;
    then B: x ∈ X ∩ Y by T1;
    thus thesis by B, BOOLE:1;
  end;
```

式 **thesis** はここでは文 **X ∩ Y <>** ∅ をあらわします。　この場合シングル・アサンプション

```
assume x ∈ X & x ∈ Y;
```

をコレクティブ・アサンプション（collective assumption）

```
assume that A1:x ∈ X and A2: x ∈ Y;
```

に置き換えると linking ができなくなるので、これは勧められません。（*10. 規約（conventions）を見てください*）

戦術２－間接法

最初の２つのステップは仮定で：

assume *the-antecedent-of-the-implication*;
assume that *the-consequent-of-the-implication*;

最後のステップは表現

thus contradiction;

です。
以下の証明は正しくありません：

```
α implies β
  proof
   assume not β ;
   .....
   thus not α ;
```

```
end;
```

なぜ正しくないのでしょう？　Mizar は命題

```
α implies β
```

を文

```
not (α & not β)
```

に変形し証明を正確に期待します。　しかしすでに

```
not β implies not α
```

は証明されており、それは Mizar が

```
not (not β & α)
```

という文に変形しています。　証明のスケルトンの正確さをチェックするシステム・モジュールである **REASONER** が用いる意味相関部（semantic correlates）にとって論理積は可換（commutative）ではありません。　これが十分な justification が欠落しているというエラーをシステムがレポートする理由です。より正確な説明には Mizar の意味論（semantics）へのより深い洞察が必要でしょう。

上の証明は以下のように*救済*（can be *saved*）できます。

戦術　3－diffuse statement

```
α implies β
 proof
   now assume not β;
   .........
   thus not α;
   end;
   hence thesis;
end;
```

now assume not β; ..... thus not α; end; という構文は *diffuse statement* と呼ばれています（現在では：**now** による証明）。　このステートメントはその命題が明示的に書かれていないという事実で特筆されますが、**REASONER**（証明のスケルトンに責任があるプロセッサ（言語処理系）のモジュール）が自分で diffuse statement の命題を再構築します。　上の diffuse statement は：

not β implies not α

という内容をもちます。　Mizar（あるいはより正確には **REASONER**）は

α implies β

という文を期待します。　しかし diffuse statement の後の **hence thesis** という linking の使用は証明中の定理の直接的 justification となります。

実際、 **CHECKER**（直接的 justification に責任のあるプロセッサ（言語処理系）のモジュール）は命題計算（propositional calculus）の恒真命題を参照して（それはいつも“それ自身で”行われるのでしょうが）証明された diffuse statement を容易に主命題に組み換えるでしょう。

８．３　同値の証明

戦術　1

命題が同値命題

α iff β

なら

α implies β　と　**β implies α**

という２つの含意命題を証明しなければなりません。　ここで２つの含意命題の列挙の順番は本質的で、入れ替えてはいけません。　かくして命題の証明は次のようになります。

```
proof
  thus α implies β proof … end;
  thus thesis proof … end;
end;
```

ここで語 **thesis** が文 **β implies α** のことを意味するのは自明です。

戦術　2－二つの diffuse statements

```
α iff β
 proof
  A: now assume α;
  .....
  thus β;
 end;
```

```
    B: now assume β;
    .....
    thus α;
   end;
  thus thesis by A,B;
end;
```

上の証明で命題 **A** と命題 **B** を持つ diffuse statement が証明される順番は本質的ではありません(inessential)。 なぜなら **CHECKER** は直接的 justification **thus thesis by A, B** を立証するときに、命題計算の恒真命題を利用するでしょうから。 さらにラベル **B:** は余計であることに注意してください。 なぜなら２番目の diffuse statement の命題を **hence thesis by A** という linking により参照することもできるからです。

８．４ 論理和（disjunction）の証明

戦術 1

命題が

$$D_1 \text{ or } D_2$$

という形をしているなら、論理和の１つの構成要素の否定を仮定し、もう一方を証明すると便利です。 その場合証明は：

proof assume not D_1**;** ... **;** **thus** D_2**;** ... **;** **end;**

といった形になります。
例えば

```
{x} \ X = ∅ or {x} \ X = {x}
  proof
    assume {x} \ X <> ∅;
    then B: not x ∈ X by Th24;
    thus thesis by B, Th22;
  end;
```

で、語 **thesis** はここでは文 **{x} \ X = {x}** を表します。 ここでもまた、ラベルは余計です； **hence thesis by Th22** という linking でラベル **B:** を持つ文を参照できますから。

戦術　2　－論理和の二つの含意命題による置き換え

γ を補助的な文としましょう。

```
α or β
  proof
    A: γ implies α proof … end;
    B: not γ implies β proof … end;
    thus thesis by A,B;
  end;
```

ラベル **B:** は linking、**hence thesis by B** を使えば削除することもできます。

戦術　3　－間接

```
 α or β
   proof
     assume that A: not α and B: not β;
      ……
   thus contradiction;
  end;
```

８．５　全称（命題）文の証明

全称ステートメントの証明をしたいときは与えられた型（type）のあらゆる（any）オブジェクトを考慮します。 Mizar ではこんな具合に進めます。 文の証明が

```
for x being θ holds … ;
```

という形のときは、いわゆる一般化（generalization）、つまり

```
let x be θ ;
```

という表現で始めます。 証明される文が

```
for x being θ st ω holds … ;
```

という形のときは、証明の最初のステップは

```
let x be θ such that L: ω ;
```

という具合になるでしょう。

例

```
for X,Y,Z being set holds X c= Y & Y c= Z implies X c= Z
  proof
    let X,Y,Z be set;
    assume A: X c= Y;
    assume B: Y c= Z;
    C: for x holds x ∈ X implies x ∈ Z
      proof
        let x such that D: x ∈ X;
        ex ∈  Y  by A,D,BOOLE:5;
        thus x ∈ Z by B,E,BOOLE:5;
      end;
    thus X c=Z by C,BOOLE:5;
end;
```

同じ証明をより簡単な方法で記述することもできます。

```
X c= Y & Y c= Z implies X c= Z
  proof
    assume A: X c= Y;
    assume B: Y c= Z;
    for x holds x ∈ X implies x ∈ Z
      proof
        let x such that D: x ∈ X;
        x ∈ Y by A,D,BOOLE:5;
        hence x ∈ Z by B,BOOLE:5;
      end;
    hence thesis by BOOLE:5;
end;
```

実際：

a) 文

```
X c= Y & Y c= Z implies X c= Z;
```

と

```
for X,Y,Z being set holds X c= Y & Y c= Z implies X c= Z;
```

はシステムにより同値として取り扱われます。　意味相関部（semantic

correlates）に渡されたときにシステムが失われた量化子を付け加えます。

b）主命題から文 **A:** と文 **B:** を*切り落とした*後、語 **thesis** は証明の最後から２番目（last but one）のステップの後項（consequent）だけを意味します。　それは明示的に書かれてはない文 **X c= Z** のことです。

８．６　存在（命題）文の証明

存在（命題）文（existential sentence）

```
ex x st W(x)
```

の証明は条件 **W** を満たすオブジェクトを指示することで構成されます。　したがってこれについての（relevant）証明は以下に示すように：

```
proof
  .....
  A: W(c);
  take x = c;
  thus thesis by A;
end;
```

となります。　シンボル **c** は今問題にしているオブジェクトを表します。　表現 **take x = c;** はそのオブジェクトを指します。　しかしどうやってそのオブジェクトを見つけるのでしょう。　そのオブジェクトを見つける方法を２つ指摘することができます。

第１は、もしその理論がなんらかの定数を持っているか、あるいは定数の定義ができるなら、考えている理論の定数から定数 **c** を構成することです。　この方法には effective という術語を使ってみましょう。

第２は、non-effective で、仮定された、あるいは前に証明された存在（命題）文を利用することです。

effective proof の例：

```
ex x,y,z being Nat st x <> y & x <> z & y <> z
proof
  take x = 0, y = 1, z = 2;
  thus thesis;
end;
```

ここで語 **thesis** は文 **0 <> 1 & 0 <> 2 & 1 <> 2** のことで、それは Mizar

にとっては自明です。

non-effective proof の例：

Π を任意の２項述語（binary predicate）のシンボルとしましょう。

```
(ex x st for y holds x Π y) implies (for y ex x st x Π y)
proof
  assume ex x st for y holds x Π y;
  then consider x such that A: for y holds x Π y;
  thus thesis
    proof
      let y;
      B: x Π y by A;
      hence thesis;
    end;
  end;
```

consider の後の linking は許されないのでラベル **A:** を削除することはできません。　仮定文が主命題から含意命題の前項（antecedent）を切り落としてしまっているので、最初に出現した語 **thesis** は文 **for y ex x st x Π y** を意味します。　二度目の出現（入れ子になった証明）する語 **thesis** は、一般化 **let y** の後で文 **ex x st x Π y** を意味します。　**CHECKER** は具体性からの抽出法則（law of abstraction from concreteness）と呼ばれる規則を適用するので、その文（訳注：２度目の **thesis**）は linking により文 **B:** から理解する（follows）ことができます（以下の *１４．チェッカーの基本的情報*を見てください）。

警告の例！

次の２つの文を注意深く読んでみましょう：

```
ex x being θ st α;
ex x being Any st (x is θ & α);
```

述語計算（predicate calculus）におけるそれらの同値性は自明で、いかなる疑問もありません。　２番目の文のカッコは余計です、しかしそれらはそのままのほうが良いでしょう。　なぜなら警告を受けた読者が誤った場所でパズルの解答を探すことになりかねませんから。　そのパズルは：Mizar の視点から見てこの２つの文は同値でしょうか？　と言う事です。　さて、そうではありません。　最初の文の証明のときに、例証 **take x;** を使います、ここで **x** は θ

という型を持ち、文 α を証明します。　２番目の文を証明するときは例証 **take x;** を使い、ここで **x** は型 **Any** を持ち、われわれは２つの文： **x is** θ と α を証明します。

９．型の変更（Change of type）

実際面では、推論の中でオブジェクトの型変更が必要になることがしばしば起きます。　これは **reconsider** という構文、すなわち型変更のステートメントの中で行うことができ、それは：

reconsider x = τ **as** θ **by …;**

あるいは

reconsider x = τ **as** θ **from …;**

という形をとります。　ここで τ は項です。
reconsider という構文は新しい変数 **x** を導入し、それは τ と同じものを意味しますが θ の型を持ちます。　型変更ステートメントは修飾する式（qualifying formula）

τ **is** θ

で justify されます。

例

```
let R be Subset of X;
A: R c= X by … ;
X = [:X1, X2:] by … ;
then consider R' = R as Relation of X1, X2 by A … ;
```

reconsider という方法ではローカルに、つまり推論（reasoning）の中で、型変更をすることができます。　グローバルな型変更は再定義で行えます（*25. 再定義*を見て下さい）。 項 τ は変数 **x** であり、型 θ を持つとしましょう。*型変更のステートメント*は：

reconsider x as θ' **by …;**

あるいは

reconsider x as θ' **from …;**

と記述されるでしょう。

Mizar 文法に従って*型変更のステートメント*という名前が用いられることになっていますが、この場合ステートメントの“働き”は変数 **x** は古い型 θ を失うことなく新しい型 θ' を獲得するので、この名前は誤解を招きやすいようです。 実際には一つの型をもう一つの型に変えるのではなく、与えられた変数に属する型の集合の拡大を行わねばなりません。 上に示した*型の集合の拡大*という操作は変数についてのみ“働き”ます。 恣意的に任意の項に作用することはありません。

１０．規約（Conventions）

１０．１ 予約

予約は式の短縮を可能にします。 表現

```
reserve x for θ;
```

は変数 **x** は θ 型を持つことを確立（establish）します。 その変数が文中に出現するときにはもう明示的な変数 x の型についての記述はされません。

```
reserve x for set;
A: for x st x <> ∅ ex a being Any st a ∈ x;
B: for x being Any holds x = x;
```

文 **A:** においては変数 **x** は **set** 型を持つと予約部に記述されています（変数 **x** は暗黙のうちに限定修飾（qualified）されています）。 文 **B:** においては変数 **x** は **Any** という型を持ちます(ここで **x** は明示的に修飾されています)。文 **B:** において型 **Any** を変数 **x** に帰属させること（ascribing to）は最初の予約文による型をキャンセルしません（訳注：型 set も残ります）。 実際、これに続く文

```
C: ex x st x = ∅;
```

においては、システムは **x** を **set** 型の変数として認識します。

１０．２ **then** による linking

then という語が文 Z に先行するということは、Z の justification において文 Z の直前にある文 **A:** を利用できるようにするということです。 この方法による justification は linking と呼ばれます。

linking では表現 **A:** が文である必要があることを強調しておかねばなりません。 したがって **proof** の後の linking の使用は許されません。 そしてコレクティブ・アサンプション（collective assumption）の後の linking も許されません。 この場合は、いくつかある仮定のどれを参照するかを明確にできないからです。 最後に、文の形を残さない構文 ― 以下に示します ― の後の linking も許されません。

- 一般化 **let … ;** の後
- 例証 **take … ;** の後
- 選択のステートメント **consider … ;** の後
- 存在仮定 **given … ;** の後
- 型変更のステートメント **reconsider … ;** の後

結論 **thus … ; hence … ;** の後の linking は許されます。

１０．３ **hence** による linking

結論の前提の一つがその結論に先行する文であるときは **thus** を**hence** に換えることで linking を使うことができます。 比喩的に言うと

then と **thus** で **hence** と同等（equal）です。

１０．４ 存在仮定（Existential assumption）

構文

```
given … such that … ;
```

は*存在仮定（existential assumption）*と呼ばれます。 これは、あるオブジェクトの存在についての仮定（**assume ex …** ）を、それらのオブジェクトのうちのひとつを選択すること（**consider …**）との組み合わせに置き換えます。 存在仮定のあとの linking は許されません。

例

ステートメントが乗法の可換性を表現しているとしましょう（suppose）。

```
(ex y st x•y = x) implies (ex y st y•x = x)
   proof
     given x such that A: x•y = x;
     thus thesis by A, T3;
   end;
```

１１．　入れ子になった証明（Nested proofs）

Mizar は他の **proof** の内側での **proof** 構文を許し、それは入れ子になった証明とよばれます。　内側の証明の構文規則は（訳注：外側の）主証明とまったく同じです。

１２．ディフューズ ステートメント（Diffuse statement）

まだ公然に（openly* 訳注）定式化されていない定理を証明するということがしばしばあります。　Mizar では、そういう構文は *diffuse statement* と呼ばれ、語 **now** で始めます。　それは：

L:　now … thus β **;　end;**

のように書かれます。　**now** という語には結論 β の証明が続きます（**now** の後には **proof** という語は書かれません）。

例

L:　now assume α **;　… take x;　thus** β **; end;**

L による直接証明は文

α **implies ex x st** β

となります。

注１

結論 β は公然に*（openly:*訳注）書かれていなければなりませんし、**thesis** という語で置き換えられてもいけません。　何故かを説明しましょう。

ラベル **L:** への参照は **now** 構文により証明された命題への参照、あるいは、より正確にはそういう推論（reasoning）の結果への参照です。　もちろん、その推論の結果が計算できるならば、 β は公然に（openly*：*訳注）（thus thesis で）定式化できるに違いありません。

注２

now の前には、**then**、**thus**、そして **hence** などのシンボルは許されません。

*訳注：openly 、openly が open form（p.20 参照）を意味するならば、openly でない文とは thus thesis を使用しないで書かれた定理という意味か？

diffuse statements についての短いコメントの結論として、この構文の使用に関係するある危険について読者の注意を喚起しようと思います。　以下の推論を検討してみてください。

```
X <> y implies γ
  proof
    assume x <> y;
    then E: x < y or y < x by AXIOMS:21;
    A: now
       assume x < y;
       ..........
       thus γ;
      end;
   now
      assume y < x;
    hence γ by A;
   end;
  hence thesis by A,E;
end;
```

上の推論は正しくありません。　ラベル **A:** の下に文

```
    x < y implies γ;
```

があります。　その文では **x, y** は定数です。　それゆえ、その文からは

```
    for x, y holds x < y implies γ;
```

に続きません。　そこで、また文

```
     y < x implies γ;
```

も続きません。　ですから結論の justification

```
    hence γ by A;
```

は誤りです。　以下に上の証明の正しいバージョンを示します：

```
x <> y implies γ
proof
  assume x <> y;
  then E: x < y or y < x by AXIOMS:21;
```

```
  A: now
     let a,b;
     assume a < b;
     .....
     thus γ;
    end;
  thus thesis by A,E;
end;
```

文 **A:** は次の全称（命題）文です。

```
     for x, y holds x < y implies γ;
```

そしてなぜそれが下の結論に適用できるのかについての理由です。

```
     thus thesis by A, E;
```

１３．反復等号（Iterative equality）

*反復等号（iterative equality）*は次の形の構文を持ちます。

τ_0 = τ_1 *justification*
.= τ_2 *justification*
.....
.= τ_n *justification;*

ここでτ_0, ... , τ_n (//?? τ_1-> τ_n) は項です。

例

```
x+y = (the add of G_Real).(x,y) by ADD
   .= x'+ y' by AR,A
   .= y'+ x' by REAL_1:2
   .= addreal.(y'+x') by AR
     .= y + x by ADD,0,A;
```

反復等号の内容とはその構文の中であらわれる最初の項と最後の項の等号です。

１４．チェッカーの基本的情報（Basic informations of Checker）

（訳注：通常informationはuncountableでcountableは「(駅の)案内所」などに限られます。 by A.S. Hornby）

1．**CHECKER** はあらゆる（every）命題計算の恒真命題を justification なしで受け入れます。

2．量化子で始まる文には justification が必要です。

3．justification の正当性を認証するときには、**CHECKER** は次の前提だけを考慮します。

- **by** あるいは **from** の後に指定されたもの、
- linking（**then, hence**）の結果であるもの。

4．等式（relation of equality）は **CHECKER** により反射律、対称律、推移律、外延律として扱われます。　例えば外延性（extensionality）により、**CHECKER** は次の文を受け入れます。

```
X = y & x <> ∅ implies y <> ∅;
```

5．**CHECKER** は、通常その前提の中に複数の全称（命題）文を持つ justification を受けいれません。　しかし **CHECKER** は O_1 が C_1 の justification で O_2 が C_2 の justification ならば

```
C1 & C2 by O1, O2 ;
```

を受け入れるでしょう。

6．**CHECKER** は自動的に、すなわち justification への参照無しで、以下の２つの古典的関数計算法の推論規則を適用します：

R1: 具体性からの抽象の法則。
R2: 一般性から特殊への遷移法則。

c が θ 型の定数を表すとすれば、上の規則は Mizar では以下のように記述できます。

規則 R1:

```
          W(c);
then ex x being θ st W(x);
```

規則 R2:

```
for x being θ holds W(x);
then W(x);
```

１５．スキーム（Schemes）

スキームは２階の文（sentences of second order）です。　最初は予約語 **scheme** とスキームの識別子から始めます。　次に（角カッコに入った）スキームのパラメータ（functorあるいは述語）のリストで、これはスキームのなかで２階（second order）の変数の役割をします。　次にコロンの後に引き続いてスキームの命題と、ことによると、前提（すなわち、スキームの仮定）がきます。　最後にスキームの justification が来ます。

以下にスキームの例を挙げます（ドットで置き換えられた justification を持ちます、訳注：該当箇所？）、すなわち前提なしの **Separation** と前提を２つ持つ **FuncEx** です。

F をプライベートな functor の識別子とし（*3. 項*を見て下さい）、そして **P** をプライベートな述語の識別子としましょう（*4. 原子式*を見て下さい）。

```
scheme Separation { F() -> set, P[Any] };
ex X st for x holds x ∈ X iff x ∈ F() & P[x] justification;

scheme FuncEx { F() -> set, P[Any,Any] };
ex f st dom f = F() & for x st x ∈ F() holds P[x, f.x] provided
A1: for x,y1,y2 st x∈F()& p[x,y1] & p[x,y2] holds y1 = y2
and
A2: for x st x ∈ F() ex y st P[x,y] justification;
```

前提 **A1:**, **A2:** には **provided** という語が先行していて **and** という分離子でお互いに分離されています。

最後にもうひとつの例、中等学校の生徒（secondary school pupils）なら必ず知っている、帰納法のスキームを：

```
scheme Ind { P[Nat] } : for k holds P[k] provided
    B1: P[0]
    and
    B2: for k st P[k] holds P[k+1]  justification;
```

多分、すでに読者は与えられたスキームの適用されるスコープは後のほうのパラメータにより決定されているのに気がついているでしょう。　例えば、帰納法のスキームは

```
for k being Nat holds φ(k)
```

という形の文の証明を許します。　ここで **φ** は1つの自由変数 **k** を持つ任意（any）の文です。　この文の justification は以下のとおり(//??a->as)。　最初に文：

```
C1:  φ(0);
C2:  for k st φ(k) holds φ(k + 1);
```

を証明します。　これらの文はスキーム **Ind:**

```
for k being Nat holds φ(k) from Ind(C1,C2);
```

を利用するための前提（premise）です。
帰納法のスキームはアーティクル **NAT_1** ですでに証明されています。　このスキームを利用するためにはそれ（訳注：**NAT_1**）を環境に付け加えなければなりませんが、それは語 **environ** と **begin** の間に指令 **schemes NAT_1** を書かなければならないということです。

定　義

１６．定義のレビュー（Review of definitions）

Mizar 言語では、標準的な数学と同様に関係（厳密には関係の名前、すなわち述語）と関数（厳密には関数の名前、すなわち functor）を定義します。　さらに、Mizar では他のオブジェクトを次々に定義するために使う（群（group）の構造体などのような）構造体を導入するための定義を使いますが、数学で通常行うよりも、さらに明示的に（explicitely）行います。　われわれの例ではそれらは群ということになるでしょう。　他方、Mizar 独特の変わった点（peculiarity）といえば、これもまた定義された型ということになるでしょう。Mizar は型のシステムを基礎としています。

述語と構造体の定義は、その正当性の検証を必要としません。　functor と型の定義は存在条件の証明を必要とし、functor のそれは唯一性条件の証明も必要です。

Mizar 言語においては、基本的オブジェクトは型、述語、そして functor です。　それらのオブジェクトのあるもの　― それらを一次オブジェクト（primary）と呼びましょう ―　はシステムにすでに組み込まれています。 それらの完全な記述は **HIDDEN**、**TARSKI**、そして **AXIOMS** のアーティクルの中で見つけられるはずです。　組み込みの型についてのレビューは*型*に関する章で見つけられるはずです。

二階（secondary）のオブジェクトは定義により導入されます。
定義されたオブジェクトはプライベートとパブリックに分類されます。　プライベートな定義（変数、プライベート述語、パブリック functor など）は有用な記数法（notation）を導入しますが、それらのスコープはローカルで単一のアーティクル内に制限されます。　本テキストでは変数の定義だけを議論します。

１７．変数の定義（Definition of variable）

構文

set x = τ ;

は変数 **x** を、項 **τ** の記号表示（designation）として導入します、 ここで **x** は変数の識別子、 **τ** は項です。

例として識別子 **X** で指示される変数の定義で始まる定理の冒頭部分を挙げ

ましょう。　読者はこの変数が他にも現れるのを予期して下さい。

```
theorem Th17:  h•(g•f) = (h•g)•f
  proof
    set X = dom(h•(g•f));
    x ∈ X iff x ∈ dom((h•g)•f);
      proof
        thus x ∈ X implies x ∈ ((h•g)•f)
        proof assume x ∈ X;
        ……
```

１８．パブリック定義（Public definitions）

さて、パブリック定義の議論へすすみましょう。　型の定義から述語と functor の定義へすすみます。

型は

- 構造体の定義、
- モードの定義、
- **is** を使う定義、すでに存在する型の略語型（abbreviations）を形成します、
- クラスタの登録。

により導入されます。　これらの４種類の型の定義を順番に議論しましょう。

１９．構造体の定義（Definition of structure）

Mizar では構造体は固定された型（fixed type）のオブジェクトの有限列で、*その特定の構造体の選択子（selector）*と呼ばれます。　構造体の定義はいかなる条件も正当性も要求しません。　特に、存在条件は Mizar 言語であるという理由で justification する必要はありません。

構造体は：

- パラメータなしの構造体、そして
- パラメータありの構造体

に分類できます。　パラメータ無しの構造体は以下のように定義されます：

struct Σ ≪ σ$_1$-> θ$_1$, … , σ$_p$-> θ$_p$ ≫;

ここで

Σ は構造体のシンボル、
σ$_1$, … , σ$_p$ は選択子のシンボルで、
θ$_1$, … , θ$_p$ は型です。

構造体のシンボルと選択子のそれはボキャブラリに含まれていなければなりませんし、それぞれ分類子*（classifiers） **G** と **U** が先行しなければなりません。（*訳注）

例１

```
struct IncStruct ≪ Points, Lines, Planes -> DOMAIN,
  Inc1 -> (Relation of the Points, the Lines),
  Inc2 -> (Relation of the Points, the Planes),
  Inc3 -> (Relation of the Lines, the Planes) ≫
```

定義された構造体の“サービス”を容易にする一連の定義を持つ構造体の定義に従うのは便利です。 上記の例を採用した、このアーティクルの著者は、構造体の定義の後に、以下に示す予約を導入します：

```
reserve S for IncStr;
```

次に彼はモードを定義します：

```
POINT of S, LINES of S, PLANES of S
```

そして public の述語：

```
A on L, A on P, L on P
```

を定義します。 これは予約によって設定された構造体 **S** に対する選択子 **Inc1**、**Inc2**、**Inc3** に対応します。 通常、構造体は与えられた理論の公理を満たす構造体のモードの母型になるためにだけ、定義されます。 考察中の例は **INCSP_1** *(Axiom of Incidency)* というアーティクルから採用されたもので、発生率空間（incidency spaces）**IncSpace** です。 読者が、良く知られた地図の断片を埋めるようなやり方で Mizar 言語に慣れることができるように、そのアーティクルを調べることをアドバイスします。

*訳注：分類子（classifier）、BonarskaやMizar in a Nutshellでは修飾子（qualifiers）と記述され、本稿でも６１ページに「２９．２ 修飾子（Qualifiers）」の項がありclassifierとqualifierの差異は不明。

上の例はパラメータを持たない構造体の定義の例でした。 パラメータを持つ構造体の定義を利用するほうがより便利な状況もあります。 パラメータを持つ構造体は次のように定義されます。

```
definition let x1 be θ1, ... , xn be θn;
    struct Σ over xi1, ... , xik
        ≪ σ1 -> θ'1(x1, ... , xn), ... , σp -> θ'p(x1, ... , xn) ≫;
end;
```

ここで、 $x_1, \ldots, x_n \in \{x_1, \ldots, x_n\}$、そして残りの変数は再構築可能なものです（*25. 再定義*の章の置換（permutation）の重ね合わせ（superposition）の例を見てください）。

例２

```
definition let F be FieldStr;
struct VectrSpStr over F ≪ carrier -> AbGroup, mult ->
  Function of [:the carrier of F, the carrier of the carrier:]
           the carrier of the carrier ≫;
end;
```

構造体のシンボルのあとに **VectrSpStr**（//??VectSpStr->VectrSpStr)、予約語 **over**、そしてその後に複数のパラメータ（この場合考えているのは、たった一つで、それは **F** です）が続きます。

２０．モードの定義（Definition of mode ）

以下に最も良く使われるモードの定義の形を示します：

```
definition let x1 be θ1, ... , xn be θn;
  mode Γ of xi1, ... , xik -> θ (x1, ... , xn) means
  :L: δ (x1, ... , xn, it);
  existence ... ;
end;
```

Γ はモードのシンボルでボキャブラリに含まれている必要があり、**M** という分類子*（classifier）が先行します。

覚え書き（NOTE）： 繰り返して現れる構文（recurrent constructions）はしばしばドットで置き換えられます。 いまの場合は語 **existence** の後の

justification はそれで置き換えられています。
上のモードの定義では、語 **existence** で指定される存在条件だけが要求されます。 存在条件は以下の定理の justification を含んでないといけません：

ex x being θ **(**x_1**, … ,** x_n**) st** δ **(**x_1**, … ,** x_n**, x);**

ここに定義定理（definitional theorem）の形をした：

for x_1 **being** θ_1**, … ,** x_n **being** θ **n, y being** θ **(**x_1**, … ,** x_n**) holds**
y is Γ **of** x_{i1}**, … ,** x_{ik} **iff** δ **(**x_1**, … ,** x_n**);**

を示します。

例

```
definition
 mode Function -> set means
:FUNC:   (for p st p ∈ it ex x,y st [x,y] = p) &
      (for x,y1,y2 st [x,y1] ∈ it iff [x,y2] ∈ it holds y1=y2);
 existence … ;
end;;
```

Function という語は導入される型を示すシンボルです。 function 型（type function）は **set** 型まで拡大します。 あるいは、そういう表現が好みならば **set** 型はfunction 型の母型（mother type）です。 **means** という語には **:FUNC:**というラベルを持つ definiens（定義文）が続きます。 **it** という **set** 型のオブジェクトは、それが順序対の集合

for p st p ∈ it ex x, y st [x,y] = p;

ならば function 型を持ち、その対の二番目の要素の唯一性（uniqueness）の条件

for x,y1,y2 st [x,y1] ∈ it iff [x,y2] ∈ it holds y1 = y2;

を満たします。
語 **existence** には定義される型が空でない（non-emptiness）という justification が続きます。 システムは以下の文の justification を期待します。

```
ex F being set st
 (for p st p ∈ F ex x,y st [x,y] = p) &
 (for x,y1,y2 st [x,y1] ∈ F iff [x,y2] ∈ F holds y1 = y2);
```

その justification では（*8. 定理の証明と存在（命題）文の証明の戦術*を見てください）空集合を **set** 型のオブジェクト：

```
take ∅;
```

とみなすことを許し（suffices）ます。

上の定義は以下の定義定理（definitional theorem）：

```
for F being set holds F is Function iff
 (for p st p ∈ F ex x, y st [x, y] = p) &
 (for x, y1,y2 st [x, y1] ∈ F iff [x, y2] ∈ F holds y1 = y2);
```

のなかに類似部分（analogue）を持ちます。　定義定理はシステムにより自動的に生成されます。　定義定理を利用するには、それに関するアーティクルのアブストラクトで、その定義定理の番号を見つける必要があります。

２１．略語（"is"を使う定義）（Abbreviations（definitions with "is" ））

さて、他のモードと等しいモードを外延的（extensionally）に定義するとても便利な方法について書きましょう。

```
definition let x1 be θ1, … , xn be θn ;
 mode Γ of xi1, … , xik is θ (x1, … , xn);
end;
```

Γ はモードのシンボルで分類子*（classifier） **M** が先行していて、ボキャブラリに含まれていなければなりません。

上に述べた定義にはいかなる definiens（定義文）も含みません。　それゆえ、実際に略語型を便利に導入することができる構文の役割を果たします。　**is** によるモードの定義は正当性の条件を要求しません。

システムは、与えられたアーティクルを処理する時、略語型を元の（original）型に置き換えます。　それゆえ、しばしば

is *の構文は型の拡大を一段実行（enforces）する。*

と言われます。

例

```
definition let H be Group;
  mode Endomorphism of H is Homomorphism of H, H;
end;
```

システムは **Endomorphism of H** という表現を **Homomorphism H, H** という表現に置き換えます。

警　　告

１．　**is** で定義された型（*25.再定義*を見てください）を再定義するのは誤りです。 これは上に与えられた例を参照すれば簡単に説明できます。 表現 **Homomorphism of H, H** は引数 **H** が２度現れるのでモードのパターンではありません。 **Endomorphism of H** というモードを再定義することで事実上 **Homomorphism of H, H** を再定義することができます。 こうしてモードのパターンではない表現の再定義を行っているのかも知れません。

２．　さらに **is** という構文によって再定義をすることは誤りです。

２２．属性とクラスタ（Attributes and clusters）

２２．１　属性

*属性*の定義は以下に示す形をしています：

```
definition let x1 be θ1, … , xn be θn;
  attr Δ -> θ(x1, … , xn) means δ(x1, … ,xn,it);
end;
```

ここで **Δ** は属性（attribute）のシンボルで分類子*（classifier） **V** が先行し、ボキャブラリに入っている必要があります。
型 $\theta_1, \ldots, \theta_n$ は*属性* **Δ** *のパラメータ*と呼ばれます。

属性 **Δ** の定義のなかの表現 “**-> θ(x1, … , xn)**”は、その属性が型 $\theta(x_1, \ldots, x_n)$ まで拡大される任意の型につけ加えられるかもしれないことを示します。

Θ（**Δ**）を型 $\theta(x_1, \ldots, x_n)$ にまで拡大されるすべての型の集合としましょう。 型 $\theta(x_1, \ldots, x_n)$ は*属性* **Δ** *の母型*と呼ばれるでしょう。 こうして **Θ**（**Δ**）は属性 **Δ** の母型へ拡大されるすべての型の集合となります。

τ を項とし、**υ**（τ）を項 τ の型としましょう。 表現

τ is Δ

、ここで υ（τ）∈Θ（Δ） です、は*属性式（attributive formulas）*と呼ばれます。　次の同値式：

$$\tau \text{ is } \Delta \text{ iff } \delta(x_1, \ldots, x_n, \tau)$$

が成立します。

例１

```
definition
  attr distributive -> Lattice means
    for a,b,c being Element of the carrier of it holds
       a ⊔ (b ⊓ c) = (a ⊔ b) ⊓ (a ⊔ c);
end;
```

（訳注：⊔ は集合の直和、⊓ は集合の直積を表します）

型 **Lattice** は属性配分子（attribute **distributive**）の母型です。

$$\Theta(\text{distributive})$$

を母型 **Lattice** に拡大されるすべての型の集合とします。　属性配分子（attribute distributive）は属性配分子の母型に拡大される型を持つそれらの項について定義される関数（function）で属性式（attributive formulas）の集合 A の中に値を持ちます：

$$\text{distributive}: \{\tau : \upsilon(\tau) \in \Theta(\text{distributive})\} \to A.$$

例２

```
definition
  attr non-empty -> set;
end;
```

型 **set** は属性 **non-empty** の母型です。　この例はアーティクル **HIDDEN** から採用されました。

例３

```
definition
 attr finite -> set means ex p being FinSequence st rng p = it;
end;
```

型 **set** は属性 **finite** の母型です。

２２．２　登録

属性は型を構築するのに使われます。　これは以下の例で説明しましょう。

例４

```
definition
  cluster non-empty θ;
  existence … ;
end;
```

ここで θ は θ ∈ Θ(non-empty) となる型です。

特に、**typ θ = Set-Family** をとるときは、それは型 **set** へ拡大されますが、θ ∈ Θ(non-empty) であり、**non-empty Set-Family** 型を得ます。

上の定義の存在条件は **x** が **non-empty** となる θ 型のオブジェクト **x** が存在することを述べていて、それは **x** ≠ ∅ を言うことになります（is to say）。　それらの定義は*登録 (registrations)* という術語がつかわれます。　もちろん、登録された属性に対し略語を導入するのは、ある時は便利かも知れません。　上の定義に基づき θ ∈ Θ(non-empty) のために **non-empty** θ 型を導入してみましょう。

例５

```
definition
  mode abbreviation is non-empty θ;
end;
```

特に、 **θ = set** と **θ = Subset of X** に対して下に示す略語を得ます。

```
definition
  mode DOMAIN is non-empty set;
end;
```

そして

```
definition let X be DOMAIN;
  mode SUBDOMAIN of X is non-empty Subset of X;
end;
```

型 **DOMAIN** と **SUBDOMAIN** はすでにアーティクル **HIDDEN** で導入されています。

パラメータを持つ属性の例を考えて見ましょう。

```
definition let T be TopSpace;
  attr open -> Subset of T means it ∈ topology of T;
end;
```

現時点ではパラメータを持つ属性の使用に関する一時的な制限があります：それらの定義をすることは良いのですが、パラメータを持つ属性を含む型を登録することは許されていません。

２２．３　クラスタ

*クラスタ*というのは集合 $\{\alpha_1, \ldots, \alpha_n\}$ のことで、$\alpha_1, \ldots, \alpha_n$ は属性です。　クラスタは型の構文で使われます。

例４の登録は１つ属性を持つクラスタの登録です。　そして以下に２つ属性を持つクラスタの例を書きます。

例６

```
definition
   cluster non-empty finite θ;
   existence …;
end;
```

ここで θ は $\theta \in \Theta(\text{non-empty}) \cap \Theta(\text{finite})$ となる型です。 上の登録は属性の順位（order）の変更をともなう登録と同じ効果をもちます。

```
definition
   cluster non-empty finite θ;
   existence …;
end;
```

この例はクラスタ内での属性の順位（order）がどうであってもシステムにとってはまったく意味がないことを強調するためのものです。

２２．４　属性の応用

属性は：

a) 属性を持つ型（attributed types）の構文：

non-empty θ、
distributive Λ、
non-empty finite θ、

distributive complemented Λ、

で使用されます、ここで $\theta \in \Theta$(non-empty)、$\Lambda \in \Theta$(distributive) です。　特に、属性を持つ型（attributed types）は

non-empty set,
distributive Lattice,
non-empty finite set,
distributed complemented Lattice です。

これらの属性を持つ型の一つ一つについて、略語を登録することができます。上の属性を持つ型の最初の２つは、それぞれ **DOMAIN** と **D_Lattice** に略されます。

b) 属性式の形成（formation of attributive formulas）：

τ is non-empty,
λ is distributive.
τ is non-empty finite,
λ is distributive complemented、

ここで τ、λ は $\upsilon(\tau) \in \Theta$(non-empty)、$\upsilon(\lambda) \in \Theta$(distributive) である項です。

２２．５　属性式(attributive formulas)対(versus)修飾する式(qualifying formulas)

*修飾する式*とは ξ **is** θ という形をした任意の式のことで、ξ は任意の項で、θ も任意の型であるのに注意して下さい。

読者には属性式（attributive formulas、上の *b* の点を見てください）と、新しい種類の式で、属性を持つ型で構成される式の、*修飾する式*（*qualifying formulas*、上の *a* の点を見てください）を混同しないように警告しておきます。（訳注：属性式は）：

ξ is non-empty θ,
ξ is distributive Λ,
ξ is non-empty finite θ,
ξ is distributive complemented Λで、

ここで $\theta \in \Theta$(non-empty)、$\Lambda \in \Theta$(distributive) で、ξ は任意の項です。

特に*修飾する式*は：

ξ is a non-empty set,
ξ is a distributive Lattice,
ξ is a non-empty finite set,
ξ is distributive complemented Lattice で、

ここで ξ は任意の項です。

両方の種類の式とも３つの部位を持ちます。　最初は項、２番目は is です。２種類の違う点を以下に示します：

- 属性式中の項の型は、属性の母型へ拡大されねばなりません。
  *修飾する式*の中の項は任意です。
- 属性式の３番目の部位は属性です。
  *修飾する式*の３番目の部位は型です。

２２．６　属性の順番

一般的に言って、*属性を持つ型（attributed type）*は：

$$\theta' = \alpha_1, \ldots, \alpha_n\ \theta,$$

という形を持ちます。　ここで $\alpha_1, \ldots, \alpha_n$ 属性であり、一方 θ は属性を持たない型です。　属性 $\alpha_1, \ldots, \alpha_n$ は型 θ' の属性という術語で表され、型 θ は型 θ' のコアという術語で表現されます。

明らかに、型 θ' は属性 $\alpha_1, \ldots, \alpha_n$ のクラスタの定義の後でのみ使用が許されます。

この時点で、属性を持つ型の中での属性の順番はシステムにとって無意味であることを再度強調しておかねばなりません。

属性式の一般的な形は：

$$\tau \text{ is } \alpha_1, \ldots, \alpha_n$$

です。　ここで $\alpha_1, \ldots, \alpha_n$ は属性、そして

$$\upsilon(\tau) \in \Theta(\alpha_1, \ldots, \alpha_n) = \Theta(\alpha_1) \cap \ldots \cap \Theta(\alpha_n)$$

の内容は論理積（conjunction）

$$\tau \text{ is } \alpha_1 \text{ \& } \ldots \text{ \& } \tau \text{ is } \alpha_n$$

です。

Mizar 言語での論理積の非交換性（non-commutativity）により属性式中の属性の順序の維持は不可欠です。

２２．７　登録継承の原理

K = {$\alpha_1, \ldots, \alpha_n$} をクラスタとし、P = {$\alpha_{i1}, \ldots, \alpha_{in}$} を任意のサブクラスタとしましょう。　すると P ⊆ K です。　クラスタ K の登録を行えばサブクラスタ P の登録は不要になります。　こうして有限の空でない（finite non-empty） Set-Family 型の登録後は、有限の（finite） Set-Family 型も空でない（non-empty） Set-Family 型も登録する必要はありません。　登録継承の原理は、型を持つ属性の導入を簡単にするので、とても便利です。

２３．述語の定義（Definition of predicate）

ここに最も頻回に行われる述語の定義の形があります。

```
definition let x1 be θ1, … ,xn be θn;
  pred xi1, … , xikΠxj1, … , xjl means
  :L:  δ(xi, … , xn);
end;
```

ここで Π は述語のシンボルでボキャブラリに入っている必要があり、分類子*（classifier）**R** が先行します。
述語の定義では正当性の条件は要求されません。　以下に定義定理（definitional theorem）の形を載せます。

```
for x1 being θ1, … , xn being θn holds;
  xi1, … , xikΠxj1, … , xjl iff δ(x1, … , xn);
```

例
以下に与える public の述語の定義：

```
definition let f;
  pred f is_one_to_one means
    :ONE_TO_ONE: for x1,x2 st x1 ∈ dom f & x2 ∈ dom f
     & f.x1 = f.x2 holds x1 = x2;
end;
```

は次に示す対応する定義定理：

```
  f is_one_to_one iff
     for x1,x2 st x1 ∈ dom f & x2 ∈ dom f &
```

```
f.x1 = f.x2 holds x1 = x2;
```

を持ちます。

２４．functorの定義（Definition of functor）

以下に最も良く使われるパブリック functor の定義の形：

```
definition let x1 be θ1, ..., xn be θn;
  func (xi1, ..., xik) φ(xj1, ..., xjl) -> θ (x1, ..., xn) means
  :L: δ (x1, ..., xn, it);
  existence ... ;
  uniqueness ... ;
end;
```

があります。　ここで φ は functor のシンボルでボキャブラリの中に入っていることが必要で、分類子*（classifier） o が先行します。

上のパブリック functor の定義で、２つの正当性の条件が要求されます：**existence** と **uniqueness** の正当性です。　さて δ (x1, ... ,xn, x) を単純なパブリック functor の定義の中に現れる定義項（definiens）としましょう。

存在条件は以下の定理の justification を含んでいないといけません。

```
  ex x being θ (x1, ..., xn) st δ (x1, ..., xn, x);
```

例

```
definition let f be Function;
  func graph f -> set means
  :GRAPH: f = it;
  existence;
  uniqueness;
end;
```

唯一性の条件は次の定理の justification を含んでいないといけません。

```
  for y, z being θ (x1, ..., xn) st δ (x1, ..., xn, y) &
            δ (x1, ..., xn, z) holds y = z;
```

例

```
definition let f,Y;
  func f"Y -> set means
  :INVERSE_IMAGE: for x holds x ∈ it iff x ∈ dom f & f.x ∈ Y;
```

```
 existence from Separation;
 uniqueness;
 proof
  let X1,X2 such that
   A1: x ∈ X1 iff x ∈ dom f & f.x ∈ Y and
   A2: x ∈ X2 iff x ∈ dom f & f.x ∈ Y;
 x ∈ X1 iff x ∈ X2;
   proof x ∈ X1 iff x ∈ dom f & f.x ∈ Y by A1;
    hence thesis by A2;
   end;
  hence X1 = X2 by TARSKI:2;
 end;
end;
```

そしてパブリック functor に対する定義定理の形を以下に示します。

for x_1 **being** $\theta_1, \ldots,$ x_n **being** θ_n, **y being** θ **(**$x_1, \ldots, x_n$**) holds**
y = ($x_{i1}, \ldots, x_{ik}$) φ ($x_{j1}, \ldots, x_{jl}$) **iff** δ (x_1, ... , x_n, **y**);

例
次の定義：

```
definition let f;
 func dom f -> set means
 :DOM: for x holds x ∈ it iff ex y st [x,y] ∈ graph f;
 existence … ;
 uniqueness … ;
end;
```

を考えてみましょう。　それは対応する定義定理：

```
 X = dom f iff for x holds x ∈ X iff ex y st [x,y] ∈ graph f;
```

を持ちます

２５．再定義（Redefinition）

再定義は定義の中の重要な種類の一つです。　多くの場合、再定義は仕様（specification）の変更から成り、元の母型からの縮小になります。　時々、定

義をする項（definiens、以下定義項と書きます）の同値のものへ変更もあります。　前者の場合は新しい母型が元の型へ拡大できることを示さねば（justify（//??justification->justify））ならず、後者の場合は新しいものと元の型はお互いに同値であることを示さねばなりません。　これらは再定義の正当性の条件で：仕様の変更の場合においては **correctness** のことで、定義項（definiens）の変更の場合には **compatibility** のことです。

総合的な観点からは、再定義は定義の一種です。　再定義の唯一の判別法は予約語 **redefine** で、それは **mode, pred,** あるいは **func** の前におかれます。

ここでは仕様の変更による再定義の議論に限ることにしましょう。
もう一回言わせてください：　受け入れることができる仕様の変更は*型の狭小化*からなります。　別の言葉で言うと、再定義で導入された新しい型は、元の定義で与えられた*元の型の狭小化*です。

*型の狭小化*の原理は次の：

定義はできるだけ一般的であるべきで、少しずつ狭められるべきである。

という概念の導入に関する勧告に従います。　実際的な Mizar を書く練習に限っては、その原理を遵守しないかぎり必然的に厄介なこと（essential complication）がおきる結果になるのが証明されています。

通常、仕様の変更における定義項（definiens）は省略します。　上に述べたように、そういう場合は元の定義項（definiens）は既知のもの（definiens understood）だからです。

（訳注：上の３行、訳者は理解できませんでした。経験者よりコメントをいただいたので注として追記。　― いま整数値関数 f が定義されています。　それを実数値関数 f として再定義したいとき、f の定義自体（定義項）は、既知のものとして再記述せず、関数の値が実数であることだけをcorrectness 項で justify することであろうと考えられます ―）

ここに最も頻回に使われるモードの再定義の形があります：

definition let x_1 **be** θ_1, ..., x_n **be** θ_n;
  redefine mode Γ of $x_{i1}, \ldots, x_{ik}$ **->** $\theta(x_1, \ldots, x_n)$;
  coherence *justification*;
end;

この再定義の justification で、次の文：

for x being Γ **of** $x_{i1}, \ldots, x_{ik}$ **hold x is** $\theta(x_1, \ldots, x_n)$**;**

を証明します。　そしてこれは典型的なパブリックな functor の再定義です。

definition let x_1 **be** θ_1**,** $\ldots$**,** x_n **be** θ_n**;**
  redefine func $(x_{i1}, \ldots, x_{ik})\ \varphi\ (x_{j1}, \ldots, x_{jl})$ **->** $\theta(x_1, \ldots, x_n)$**;**
  coherence *justification*;
end;

justification の中で：

$(x_{i1}, \ldots, x_{ik})\ \varphi\ (x_{j1}, \ldots, x_{jl})$ **is** $\theta(x_1, \ldots, x_n)$

を証明します。

例１

```
definition let X be set;  let f,g be Permutation of X;
 redefine func g•f  -> Permutation of X;
   coherence ::(//??cohenrence->coherence)
    proof
     g is_one_to_one & f is_one_to_one & rng g=X & rng f=X by
PERM;
     then g•f is_one_to_one & rng(g•f) = X by FUNCT_1:46,T29;
     hence g•f is Permutation of X by PERM;
   end;
end;
```

functor **g•f** のパターンの中には変数 **X** は出現しないのに注意してください。これは変数 **X** が変数 **f** あるいは変数 **g** という手段で“再現可能”であるために許されます。

データベース

２６．データベースの内容（Content of the Data Base）

Mizar システムのデータベースは **¥MIZAR** というディレクトリに系統的に構築されています。　そこには２つのサブディレクトリ:

¥DICT
¥PREL

があり、サブディレクトリ **¥DICT** には **.voc** の拡張子を持つボキャブラリ・ファイルがあります。　それらはアーティクルを作った著者により準備されたものです（あるいは、著者によりすでに存在するものを彼の必要に適合するよう変更されたものです）。

.voc の拡張子を持つファイルにはモード、述語、functors、属性、構造体、選択子（selector）、そして与えられたアーティクルで使われる予定の関数の引数（bracket）などのシンボルの宣言があります。　シンボルの宣言はシンボルそれ自身の分類子*（classifier）　から構成されます。　さらにそのファイルは functor の優先順位の指示を含みます。　この場合優先順位とは結合の強さについての情報です。

サブディレクトリ **¥PREL** には：

.nfr, .def, .dno, .dco, .the, .sch, .dcl

の拡張子を持つファイルがあります。
それらは与えられたアーティクルから **LIBRARIAN** により形成され、適切に処理されてコードにした定義（**.nfr, .def, .dno, .dco**）、定理（**.the**）、そしてスキーム（**.sch**）を含みます。　参照するアーティクルの環境の宣言中に定義、定理、スキームなどを結合する指令があれば、それらのディレクトリから適切なファイルが Mizar の **ACCOMMODATOR** により抽出されます。

ユーザは Mizar ディレクトリの中に

¥MML
¥ABSTR

などのように自分のサブディレクトリを作成します。　それらは **Main Mizar Library** (訳注：現在は Mizar Mathematical Libraryです)を形成し、文書（documents）としての役割を演じ、システムはこれを使用しません。

¥MML サブディレクトリには Mizar article を保持する **.miz** という拡張子

を持つファイルがあります。

¥ABSTR サブディレクトリには **.abs** 拡張子をもつファイルがあります。それら（訳注：のファイル）はアーティクルのアブストラクトであって、与えられたアーティクルの元のテキストから補題、プライベートなオブジェクトの定義、そしてすべての証明を除去し、定理と定義に現在使われている番号（current numbering）をつけて得られたものです。　ユーザは、自分のアーティクルにおいてそれらが必要ならば、これらの定理と定義の番号を参照しなければなりません（元のファイルの番号は、与えられたアーティクルのテキストの中だけの局所的な意味しか持ちません）。

２７．指令（Directives）

環境部（environment）には５種類の指令をおきます（訳注：６種類？）。　すなわち、

－*vocabulary directive*

vocabulary $\alpha_1, \ldots, \alpha_n$**;**

－*signature directive*

signature $\beta_1, \ldots, \beta_n$**;**

－*definition directive*

definitions $\beta_1, \ldots, \beta_n$**;**

－*theorem directive*

theorems $\beta_1, \ldots, \beta_n$**;**

－*scheme directive*

schemes $\beta_1, \ldots, \beta_n$**;**

－*cluster directive*

clusters $\beta_1, \ldots, \beta_n$**;**

ここで $\alpha_1, \ldots, \alpha_n$; はボキャブラリの名前で、$\beta_1, \ldots, \beta_n$ はアーティクルの名前です。

処理の経過中にシステムは指令の順番を監視します。　この順番は同じシンボルの定義（再定義を含む）の情報を持つ signature 指令の参照に関して本質的と思われます。　以前述べたように一つの同じシンボル（訳注：名）は何回も定義されているかもしれません。　そして拘束力のある定義（binding definition）は常に環境の中で最後に指定された signature です。　順序が正しくないと、先に書いた指令ではなく後の指令が優先してしまいます。

例
シンボル **U** は functor のシンボルです。　**A U B** を２度定義します。

（定義）１．アーティクル　α**.miz** の中で

- domains の合計を表す **DOMAIN** として

（定義）２．アーティクル　β**.miz** の中で

- sets の合計を表す **set** として

ご存知のように、**DOMAIN** 型は **set** 型に拡大されます。
すでにアーティクル　γ１**.miz** と　γ２**.miz** の中でそれぞれ **signature** α、β**;** と **signature** β**;** 指令を採用（adopted）しているとしましょう。　**A,B** は定義域（domain）としましょう。
γ１**.miz** の中の項 **A U B** は **set** 型になります。　なぜそうなるのでしょう？　**PROCESSOR** は変数 **A, B** の型を上の（定義）２．にあわせようと試みます。　それは変数 **A, B** の型を **set** 型に拡大することなので成功するのは明らかです。
γ２**.miz** の中の項 **A U B** は **DOMAIN** 型を持ちます。　なぜならプロセッサはその項に上の（定義）１．を適用できるからです。
覚書（NOTE）：項

A U B exactly DOMAIN

は　γ１**.miz** 中で **DOMAIN** 型を持ちます。

２８．(//??39->28)アーティクルの名前（Name of Article）

アーティクルのファイルは **.miz** の拡張子を持たねばならず Mizar article という術語で呼ばれます。　Mizar article の（訳注：ファイルの）名前は多くても８個の：文字、数字、接続詞（訳注：．）とアポストロフィ ′ などの記号からなります。　Mizar アーティクルの名前は数字だけでも（訳注：Mizar の）予約語であってもいけません。

注（REMARK）：使用法ですが、ファイル名の文字は大文字でないといけません。

ですから **TARSKI** はその使用法に適いますが、**tarski** や **Tarski** は適いません。

２９．ボキャブラリ（Vocabulary）

２９．１　ボキャブラリの名前

ボキャブラリは修飾子（qualifier）といっしょにシンボルを列挙します。 それはまた functor のシンボルの拘束力の強さ（binding strength）を示します。

ボキャブラリは α**.voc** ファイルを形成し α はボキャブラリ・ファイルの名前（*name-of-vocabulary-file*）です。 名前 α は、必ずというわけではありませんが、大体は Mizar article の名前です。 例えば **FUNC.voc** というファイル名は Mizar article からとられたものではなく：データベースの **FUNCT_1.miz, FUNCT_2.miz, … ,** からのものです。 しかし ― 少なくとも現在のところ ― **FUNC.MIZ** というアーティクルはありません。

２９．２　修飾子（Qualifiers）

*修飾子（qualifiers）*と呼ばれるシンボル **M,R,O,G,U,K,L,V** はそれぞれ以下のシンボルを導入するのに使われます：

M *モード（mode）*,
R *述語（predicate）*,
O *ファンクタ（functor）*,
G *構造体（structure）*,
U *選択子（selector）*,
K *ファンクタの右カッコ（right functor bracket）*,
L *ファンクタの左カッコ（left functor bracket）*,
V *属性（attribute）*.

例
下に **FUNC.voc** というボキャブラリ・ファイルがあります。

```
Ograph 128
Oid 128
O. 100
MFunction
Ris_one_to_one
```

最初のカラムには修飾子が入ります。　そして２番目のカラムからスペースまではシンボルが入ります。　スペースの後には拘束力の強さ、それは優先度とも呼ばれますが、を示す数があります。　これは functor のシンボルにだけ適用されます。　与えられた functor の拘束力の強さはシンボルに続く数字であらわされます。

さて **graph, id, .** は functor のシンボルです。　**Function** はモードのシンボルです。　そして **is_one_to_one** は述語のシンボルです。　**128** と **100**という数字は優先度の重要性(magnitude)を表します。 優先度は functor シンボルの場合にのみ示されます；　かくして functor シンボル **graph** と **id** は優先度 **128** を持ち、**.** は優先度 **100** を持ちます。

注（REMARK）

１．　述語シンボルの拘束力の強さは functor シンボルよりも常に弱く拘束するので、表示してありません。

２．　与えられた functor シンボルの優先度を表す数字は少なくとも１つ分のスペースだけそのシンボルから離れている必要があります。

３．　修飾子と対応するボキャブラリ・シンボルの間には１つのスペースも無いことがあります。

２９．３　拘束の強さ

拘束の強さを表す数字が大きいほどその functor シンボルの拘束力も強くなります。　functor の優先度は自然数の 0 から 255 の間です。　ある種の functor シンボルの場合にはその優先度を示す数字を持たないものがあります。その時はそのシンボルは標準的優先度を持ちます。 標準的な拘束力の強さ（優先度）は 64 です。

functor シンボルの拘束力の強さの概念は与えられた拘束力の強さを持つ操作（operation）の一連の続きの順序と関連しています。　ある種のシンボルの優先度を記憶していることは、少なくとも、余計なカッコの除去には大いに役立つでしょう。

第二部： システム

３０．システムの予備的情報（Preliminary information on the system）

Mizar コンピュータシステムは Mizar 言語で形式化された定理を、より正確には、それらの定理を含む Mizar article を処理し検証します。　それは **ACCOMMODATOR** と **PROCESSOR** と呼ばれる２つの PC Mizar システムのプログラムにより行います。　**ACCOMMODATOR** (//?? ACCOMODATOR -> ACCOMMODATOR)はデータベースからそのアーティクルの環境部に書かれた指令に基づいて必要な情報を抽出し、**PROCESSOR** はアーティクルのテキストを検証（verify）します。

Mizar article は通常システムと一体化したエディタを使って書かれます。もっともポピュラーなエディタは multiEdit、あるいは Mem、そして Brief です。

今から **GEOM.miz** を書くと想像してみて下さい。　これはコマンド

accom geom. と **mizar geom**

によりそれぞれ順番にプログラム **ACCOMMODATOR** と **PROCESSOR** を呼んでいます。　それぞれのプログラムが処理を終了した後に、（訳注：プログラムが吐く）リストには、アーティクルが正しいかあるいはエラーを含むかによって、**THANKS OK** あるいは **SORRY** が出力されます。　エラーの数はテキスト中に示され、エラーの修正を容易にします。

しかし処理が中断されることもあります。　処理の中断の理由の一つはシステムの不正なインストレーション、あるいはプライベート・データベースとパブリック・データベースの不一致です。　いわゆるバグ、あるいはシステム内のエラーは、非常に稀なものですが、もう一つの理由です。　もしシステムがあなたの証明を受け入れず、あなたは受け入れるべきだと思うなら、PC Mizar システムの実装の著作者、すなわち A. Trybulec か Cz. Byliński に相談するようアドバイスします。

書いたアーティクルは連続的に証明することを推奨します。　それは現在出ているのエラーの除去を容易にし形式化を促進します。　ノートではなくコンピュータで仕事をすることも勧めます。　もちろん大きな問題は紙の上や黒板の上のほうが解決により便利でしょう。

３１．PC Mizarシステムのプログラム（Programs of the PC Mizar system）

PC Mizar システムはデータベースといくつものプログラムをカバーしますが、プログラムのうち３つは不可欠です。　すなわちこれらの：

ACCOM.exe (ACCOMMODATOR),
MIZAR.exe (PROCESSOR), そして
EXTRACT.exe (EXTRACTOR)

はシステムと密接に結びついています。
記述を簡単にするために、すべてのファイルは標準ディレクトリに置かれているものとします。

３２．　PC Mizar システムのインストレーション（Installation of the PC Mizar system）

PC Mizar システムは**IBM PC XT/AT**コンパチブルなコンピュータにインプリメントされます。　Mizar article はその種のコンピュータでのみ書くことができます。

PC Mizar システムはハードディスク上の C:¥MIZAR ディレクトリにインストールされます。　そのディレクトリはシステムの基本的プログラム、システムファイル、そして入力されたマクロのコマンドを含みます。　Main Mizar Library のデータベースファイルもそのディレクトリに置かれます。
<name-of-article> は Mizar テキストを含むファイルの名前です。　そのテキストは **.miz** の拡張子を持たないといけません。　例えば **BOOLE.miz** という具合です。　その拡張子は常に暗黙的に **ACCOMMODATOR, PROCESSOR,** そして **EXTRACTOR** によりテキストの名前に結びついています。　テキストの名前は拡張子なしに書かれます。

３３．アコモデータ（Accommodator）

ACCOMMODATORは:

accom *<name-of-article>*

で呼ばれます。

ここで手短に基本プログラム **ACCOM.exe** について書きましょう。 ACCOMMODATOR は関係するアーティクルの環境に書かれている指令に基づいてデータベースから要求された情報の項目（item）を引き出します。 こうして受け取る情報の項目は ACCOMMODATOR によって以下のファイルに記録されます：

<name-of-article>.**dic**,
<name-of-article>.**frm**,
<name-of-article>.**atr**,
<name-of-article>.**eno**,
<name-of-article>.**dfs**,
<name-of-article>.**thl**,
<name-of-article>.**ths**,
<name-of-article>.**vcl**,
<name-of-article>.**sgl**,
<name-of-article>.**ecl**

上のファイルに含まれた情報は、なかでもサマリー・ボキャブラリとフォーマットの記述、signature、定義項（definienses）、ステートメント、そしてスキームをカバーします。

その次に ACCOMMODATOR が *<name-of-article>*.**err** を作成します、それは環境部でのエラーを記録します。 そのファイルでは一連の行は、行を表す３種類の数字の組を持ち、それぞれ順番に、行番号、列番号、エラー番号を表します。 そのファイルに基づいてエラーを同定するのは、ご覧になるようにとても便利とはいえません。 それが違う方法を提案する理由です。

コマンド：

errp *<my-directory>*¥*<name-of-article>*.**miz c:¥mizar¥mizar**

に従ってそのプログラムは *<name-of-article>*.**lst**、これは今あつかっているテキストのコピーにエラーを記入したもの、いわゆるリスティング、を作ります。 さらにエラーの説明を持つファイル、**Mizar.msg** が次のコマンド：

C:¥mizar¥mizar

で結合されます。

ACCOMMODATOR を使用するときは、 **PROCESSOR** は **ACCOMMODATOR** が準備したファイルを通じてのみライブラリと通信することをいつでも心の留めておく必要があります。　かくして環境 and/or データベースの変更は新しい accommodation（訳注：調節、適応）を必要とします。

ACCOMMODATOR を呼ぶのを、環境 and/or データベースの起こる可能性のある変化だけに制限することで、アーティクルの処理を相当にスピードアップできます。　明らかにこのアドバンテージを利用するために **ACCOMMODATOR** と **PROCESSOR** はお互いに独立にインストールされ、そのため２つの違うコマンドで呼ばれることを知っている必要があります。

３４．プロセッサ（Processor）

PC Mizar のプロセッサ（とその次のいわゆる **ERRPRINTER**, それはいわゆるリスティングを形成します　－　両方の項とも後で説明します）の呼び出しは次の形：

miz <*name-of-article*>

で行われます。　テキストの処理中はいつでも **Ctrl Break** を押すことで **PROCESSOR** の動作を中断させることができます。　これは **PROCESSOR** によるテキストの解析を中断し、中断の信号としてのエラー（error #1255）が中断の発生した行に示されます。

PROCESSOR はテキストの処理中に発見されたエラーのリストを提示します。エラーのリストはインプットされたテキストと同じ名前で、拡張子 **.err** を持ち、Mizar テキストと同じディレクトリに作成されたファイルに置かれます。もし、テキストが正しければファイルは空のままです。　このエラーのファイルは３組の数字の形式でエラーの記述があります。　与えられたエラーの記述はテキスト中の位置（すなわち、行番号と列番号）とエラー番号を含みます。このエラーのファイルはいろいろな目的に使用されます。

１　エディタ（例えば Mem）あるいは他のプログラムを使って、テキスト中のその位置を見つけて番号により同定するため；

２．いわゆる **ERRPRINTER**（プログラム **ERRP**）によるリスティングを作成するため。

ERRPRINTER はテキストとエラーのファイルを読んでリスティングを作成します。こうしてリストは（リスト中に）エラーの指示を持つテキストのコピーと、テキスト中で発見されたエラーの説明をあわせた（joined）ドキュメントを形作ります。　テキストが正しければ、エラーの説明ではなく、**Thanks OK** という記述の結論を持つリスティングが表示されます。　リスティングは拡張子 **.lst** 以外はインプットされたテキストと同じ名前を持ちます。　リスティングは解析中のテキストと同じディレクトリに作成されます。

３５．エクストラクタ（Extractor）

EXTRACTOR は：

extract <*name-of-article*>

で呼び出されます。

EXTRACTOR は以下のファイルを作成します：

<*name-of-article*>**.nfr** – フォーマット、
<*name-of-article*>**.dno** – モードのパターン、述語、そして functor そしてパターンの中で発生する引数の型、
<*name-of-article*>**.def** – 定義される項（definienda）、
<*name-of-article*>**.dco** – 結果の型の記述、
<*name-of-article*>**.sch** – スキームのリストとその記述、
<*name-of-article*>**.the** – 定理のリストとその記述、
<*name-of-article*>**.dcl** – クラスタのリスト。

これらのファイルは **¥PREL** に転送されるはずです。

３６．アブストラクトの準備（Preparation of abstract）

Mizar article が正しければ、それはディレクトリ **¥ABSTR** にある **ABSTRACTOR**（*26.データベースの内容*を見てください）と呼ばれる **MIZ2ABS** プログラムを使ってアブストラクトに変換することができます。　このプログラムはまた処理中に見つかったエラーのファイルも作成します。

ABSTRACTOR は Mizar article をそのアブストラクトに変換します。　アブストラクトはアーティクルと同じ名前で、拡張子 **.abs** を持ちます。　そのファイルはあつかっているテキストと同じディレクトリに作成されます。　アブス

トラクトは実際にはアーティクルのパブリック・オブジェクトの定義と定理を含む部分のコピーです。　補題、証明、そしてプライベート・オブジェクトの定義は無視されます。

３７．正しいアーティクルの使用（Use of correct article）

検証されたアーティクルは Main Mizar Library に提出されるかも知れません。　また、作業中のローカルライブラリに含めることもあります。　これは、次のアーティクルを書く時に便利ですし、多くの場合不可欠です：通常、われわれのアーティクルは主題に関して相互に関連しており、以前のアーティクルのそのままの延長とは言えないのです。　この目的のために

１．データベースのファイルを作成すること。
２．データベースのファイルをローカル・データベースに転送すること。

をしなければなりません。　さらにアブストラクトの準備は十分やる価値のある事柄です。

３８．ユーザへの実用的なアドバイス（Practical advice for users）

１．基礎的な定義と基礎的な定理（すなわち*ステートメント*と*スキーム*です）から書き始めてください。

２．ボキャブラリを作り、その中にシンボルとそれらの修飾子を書いてください。　この作業の時に、できる限りそれぞれの優先度をボキャブラリに書いて下さい。

３．指令を書いてください。

４．予約を追加してください。

５．それらの定義の正当性条件を justify するのに有用と思われる定理を定義の前において下さい。

６．**by** と **from** すなわち、いわゆる simple justification を除いた証明のスケルトンの全体を書き出して下さい。　もし **#4** エラーだけならスケルトンは正しいのです。　その “fours（訳注： #4 error のこと）” だけが残るようになるまでがんばってエラーと闘って下さい。

7．推論の繰り返しのモジュールを削除することで証明を短くするための補助的な定理を追加してください。　Mizar では証明が完全に形式化されているので、それはやるに値する作業です。

8．証明を短縮するための、いくつかのプライベート・オブジェクト　― 変数、述語、functors ―　の導入は、やる価値があります。

9．最後に単純 justification を追加し“fours”（ **訳注：＃4**エラー）の退治を始めてください。　エラーの主な原因は一つの **by** の後にふたつの全称（命題）文を使ってしまうことです。　残念ですが証明は時に長くなければなりません　― Mizar にあまりに多くを期待しないように。　より強力な **CHECKER** をプログラムして作ることも可能かも知れませんが、待ち時間の延長というコストを払わねばなりません。

あとがき

ここでは Mizar を使うビギナーにはあまり重要ではないとおもわれる以下に示すサブジェクトは無視しました：

- *per cases* の証明、
- 定義拡張（definitional expansion）による証明、
- 複合定義（compound definitions）
- 仮定を持つ定義（definitions with assumption）
- プライベートのfunctorと述語の定義
- 理解された特性（understood properties）
- 反意語と同意語（antonyms and synonyms）

筆者は相反する感情（mixed feelings）でこの *An Outline of PC Mizar* を読者に提示します。 テキストは未熟でたぶん色々な変更が必要なように思います。しかし、たとえそのようなテキストでも Mizar の使用に有用であるのは明らかであると言われました。 この *Outline* を読むことが害よりも善をなすことを希望しています。 読者は、ぜひ全てのコメントを下記のアドレスに送るよう心からお願いします。 それは私のこの *Outline* についての更なる仕事に重要であろうと思うからです。

M. Muzalewski,
c/o Warsaw University
Bialystock Campus,
Institute of Mathematics,
ul. Akademicka 2,
15-267 Bialystok,
Poland

Bibliography

[1] Bonarska, E., An Introduction to PC Mizar, Fondation Ph. Le Hodey, Brussels 1990.

[2] Rasiowa, 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., Klasyzny rachunek kwantyfikatorów (The Classical functional Calculus), Warsaw 1970.

[5] Trybulec, A., Rudnicki, P., A Collection of TEXed Mizar Abstracts, University of Alberta, Canada, 1989.

[6] Trybulec, A., Syntaktyka Mizara (Mizar Syntactics), Bialystok 1989.

[7] Matuszewski R. (ed.), Formalized Mathematics, Vol.1, 2, 3, Fondation Philippe le Hodey, Brussels 1990, 91, 92

訳者あとがき：信州大IT大学院、しろだぬき先生のクラスで Mizar を勉強中に個人的な必要で翻訳しました。 訳者が明らかに原著の typographical error であろうと考えて修正した箇所を（//?? -> ）で示してあります。 元の表現を//??の直後に記し、->の後に変更後の表現を置きました。 読者の責任での使用、また誤訳や不十分な表現の指摘、改良のためのコメントをお願いします。 Mizar 学習のためだけの個人あるいは小グループでの使用を想定して公開します。 文責は小生にあります。 最後になりましたが Zton 氏に貴重なご意見をいただき、引用もさせていただきました。 ここに記して感謝します。

2008年 早春 l'Hospitalier

索　　引 (Index)

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

あ

い

お

ひ

ふ

へ

ほ

む

め

も

ゆ

よ

ら

り

れ

ろ