MIZAR MANUAL

by FREEK WIEDIJK

Translated by l'Hospitalier
2007

日本語版 5.0i

Permission of translation by Dr. Freek Wiedijk on Sep 16, 2007

That's fine with me.　However, it would be nice if your translation clearly states that I don't know Japanese, so that I am not able to judge whether the translation is correct.　But do whatever you like with the text, I don't mind.

Freek

私には（翻訳は）うれしいことですが、翻訳の中に明瞭に「私（Freek Wiedijk）は日本語を知らないので、翻訳が正しいのかどうか判断することができません」と書いていただけるとありがたいと思います。　しかし、気になさらず、どうぞテキストを好きなようにお使いください。

フレイク

Nomenclature （用語法）

adjective(s)→adjective(s)／形容詞、ancestor(s)→先祖型、antecedent→前項、argument(s)→引数、atomic formula(s)→原子式、associativity→結合律、conjunct→結合子、definition(s)→定義、directive(s)→指令、disjunction→論理和、equality→等式、 existential cluster(s)→存在性のクラスタ、 existential quantifier→存在性の量化子、 expression(s)→表現、 formalize→形式化、 field(s)→フィールド、formula(s)→式／公式、 functor→ファンクタ／関数記号、 identifier(s)→識別子、image(s)→像、 instantiate→インスタンスを作成、 iterative equality→反復等号、justification→ジャスティフィケーション、 mode(s)→モード、lemma(s)→補題、notation(s)→記述／記数法、 notion(s)→概念／考え、 occurrence→存在／発生、operator(s)→演算子、 predicate→述語、 proof(s)→証明／証明部、 properties→特性、 prover→証明子、quantale(s)→クオンテイル、 quantifier(s) →量化子、 radix type→基本型、redefinition(s)→再定義、 reduce→縮小する、 relation(s) →関係、reflexivity→反射律／再帰律、scheme→スキーム／枠組み、set(s)→集合、skeleton→スケルトン／骨格、 square→平方数、 step(s)→ステップ、 statement(s)→ステートメント、 structure(s)→構造体、 subproof(s)→サブプルーフ、 subset(s)→部分集合／サブセット、 symbol(s)→シンボル、 term(s)→(数式の)項、 thesis／theses→命題、 theorem(s)→定理、 token→表象、 type(s)→型、 uniquely→一義的に、universal quantifier→全称量化子／全称記号、 vocabulary(-ies)→語彙、

英文に対応する適当な日本文が見つからないと思われる場合は、（1）そのままの英文の使用、（2）カタカナ表記への変換、などをしています。　訳語の使用の適切さに確信がない場合は訳語の直後のカッコ内に英文を記載してあります。　誤訳、日本語の文法上意味上の不適切な訳、誤字、脱字、未熟な表現、難解な文章などは御指摘いただければ少しずつ改善するつもりです。　翻訳という作業の性格上、良い記述を思いついても、元の英文の意味するところを大幅に逸脱するような表現を採用することはできません。文責は l'Hospitalierにあります。　個人的なMizar学習のため以外の使用は想定されていませんので、よろしくお願いいたします。

目　次　（Table of Contents）

第１章　Writing a Mizar article in nine easy steps

「９つのやさしいステップで Mizar Article を書く」

この章を読んで練習問題をやれば､読者は人の助けを借りずに基本的な Mizar article を書くことができるようになるはずです。

この章では、Mizar article を完成するのに必要な９つのステップの説明をします。　それぞれのステップで、その内容がどのように例題に結びつくかを示してあります。　もちろん練習問題もありますので、自分の理解がどの程度進んだのかをテストすることができます。

この章の内容を理解するためには、いくつかの基礎的な数学の知識が必要ですが、あまり多くは必要ありません。

- 第１階述語論理
- 基礎的な集合論についてのある程度の知識

について勉強されると良いでしょう。 なぜなら Mizar は述語論理の最上層の集合論を基礎にして構築されているからです。

Mizar を書く時、最も難しいのは MML（訳注：Mizar Mathematical Library の略）中で、必要とする項目を探し出すことです。　残念ながらこれについては簡単な解決法がありません。　それ以外は、Mizar article を書くことは直截（straight-forward）なことであるのがおわかりいただけると思います。

１．１　Step 1:　the mathematics　「数学」

Mizar article を書く前に、まず何をしようとしているかを決めねばなりません。「すぐに article を書き始めない」ということはとても重要なことです。
ある証明の説明で　― 人がそれを読んで、宙を見つめ「ああ、なるほど！」 ― とつぶやくまでに５分かかる証明は、formalize（形式化、訳注：Mizar article を書いての証明）には約一週間かかります。　だからこそ Mizar article を書き始める前に、十分な時間をとって数学的な検討や改良をすることはとても意味があります。

例： この章のための例として **`my_mizar.miz`** というファイル名の Mizar article を載せました。　ピタゴラスの定理に関するものです。　それは３つの自然数の組 {a, b, c} が、式：

$$a^2 + b^2 = c^2$$

を満たすというものです。　ピタゴラスの定理は、三辺の長さが３つの自然数になる直角三角形と関連があります。 最もよく知られた数字の組は {3,4,5} と {5,12,13} ですが、もちろん無数の数字の組合せがあります*。

これから formalize（形式化）しようとする定理はピタゴラスの定理を満たす３つの数字の組（Pythagorean triples）は全て：

$$a = n^2 - m^2 \qquad b = 2mn \qquad c = n^2 + m^2$$

という式で与えられる数 a、b、c、 あるいはそれらの整数倍で与えられることを示しています。
例えば、{3,4,5} の組は n=2, m=1 で、{5,12,13} の組は n=3, m=2 で与えられます。

この定理の証明は直截（straight-forward）です。　a と b が互いに素である場合だけを考察する必要があります。 偶奇性（parity）についての考察は、a と b のどちらかが偶数ならば、それ以外の（b または a）の数と c は奇数であることを示しています。（*訳注：結城 浩著「数学ガール／フェルマーの最終定理」第２章の「原始ピタゴラス数は無数に存在する」の証明（と第１０章の階差数列を使った証明）など参照）　そこで、もし２つの数（a または b）のうち b が偶数ならば：

$$\left(\frac{b}{2}\right)^2 = \frac{c^2 - a^2}{4} = \left(\frac{c-a}{2}\right)\left(\frac{c+a}{2}\right)$$

が成立します。　しかし、もし２つの互いに素な数の積が平方数であれば、２つの数はいずれもある数の平方数になります。　そこで：

$$\frac{c-a}{2} = m^2 \qquad \frac{c+a}{2} = n^2$$

となります。　これが求める式です。

Exercise 1.1.1　Mizar Mathematical Library（MML）を調べて Mizar が数学とコンピュータ科学のどの領域でも同じ程度に適切になるように構築されているか、あるいは Mizar は特定の領域に最適化されているのかを判断してみてください。　後者の場合、Mizar の得意領域は、あるいは不得手な領域は何でしょう？

Exercise 1.1.2　直角三角形の辺 a、b、c は $a^2 + b^2 = c^2$ という式を満たす、というのがピタゴラスの定理であると述べました。　この定理を MML の中で探してみてください。 もし見つかれば、その article の名前とその定理は何を

参照しているか？を記してください。　見つからないか、あるいは見つかっても、そのやりかたが気にいらなければ、（この章を終了する時のための練習として）それを証明する article を自分で書いてみてください。

１．　２　　Step 2：the empty Mizar article「空のミザール文」

Mizar article を書くには、正常に動作している Mizar system とそれに入力する Mizar article のファイルが必要です。　実際、おわかりのように２つのファイルが必要で、それらは Mizar article の .miz ファイルとボキャブラリの .voc ファイルです。

このチュートリアルではファイルの命名法は Windows の流儀（conventions）に従います。　Unix 上の Mizar ではバック・スラッシュ（訳注：日本語環境では¥）の取り扱いが異なります。　例えば、**text¥my_mizar.miz** は Unix 上では **text/my_mizar.miz** となります。

すでに Mizar system が正しくインストールされていると思います。　インストールのやりかたは Mizar の配布ファイル中の **readme.txt** ファイルを見てください（Unix では **READEME** ファイルです）。

自分の article を書くには、**text** と **dict** というサブディレクトリをもつディレクトリに行く必要があります。　そのディレクトリがなければ作ってください。　**text** ディレクトリには **my_mizar.miz** という名前の空のファイルを、**dict** ディレクトリには **my_mizar.voc** という名前の空のファイルを置いてください。（ご自分で article を書いているなら、**my_mizar** のかわりにより適切な名前を使用してください）

最小の有効な **.voc** ファイルは空ファイルですが、最小の有効な **.miz** ファイルの内容は：

```
environ
begin
```

となります。　お気に入りのテキストエディタを使って **my_mizar.miz** ファイルを作り、その内容を上の２行にしてください。　それからコマンド：

```
mizf text¥my_mizar.miz
```

を入力してチェックしてください。　これで article と文法の両方がチェックされます。　もしすべてがうまく行けば：

```
Make Environment, Mizar Ver. 7.0.01 (Win32/FPC)
```

```
Copyright (c) 1990, 2004 Association of Mizar Users
-Vocabularies-Constructors-Clusters-Notation
Verifier, Mizar Ver. 7.0.01 (Win32/FPC)
Copyright (c) 1990, 2004 Association of Mizar Users

Processing: text¥my_mizar.miz
Parser [ 2 ] 0:00

Analyzer 0:00
Checker [ 1 ] 0:00
Time of mizaring: 0:00
```

という出力が得られます。　これは、この ‘article’ にはエラーがないことを示しています。（もし、emacs エディタを Mizar mode で使っている場合は、**mizf** コマンドを入力する必要はありません。　その場合はただ Ctrl-c RET と打てば良いのです。）

こんどは **text** ディレクトリに **my_mizar.miz** ファイル以外に、25 個のファイルが生成されています。　その中身を見る必要はありません。　それらのファイルは Mizar system の内部使用のためにだけ必要な物だからです。　次のステップは **.miz** ファイルと **.voc** ファイルを結合することです。　article に **vocabularies** 指令を：

```
environ
  vocabularies MY_MIZAR;
begin
```

と追加してください。（重要：**vocabulary** ファイルのファイル名は大文字だけを使ってください！　これは DOS 由来の Mizar system の名残（り）です。）

さてこれで article を書き上げる準備ができました。　さしあたり、もし article にエラーがあったとき、何が起きるか判るようにするためファイルの最後尾に：

```
environ
 vocabularies MY_MIZAR;
begin
 hello Mizar!
```

という具合に追加をしてみましょう。　再度 **mizf text¥my_mizar.miz** を走らせれば、Mizar はファイルの中にエラーメッセージを挿入します。　エラーは ***** と数字で表され、エラー箇所の下の行に現れます。　**mizf** コマンドの後

で、ファイルの内容は：

```
environ
 vocabularies  MY_MIZAR;

begin
 hello Mizar!
::> *143,321
::> 143: No implicit qualification
::> 321: Predicate symbol or "is" expected
```

となります。　もちろん「**hello Mizar!**」は有効な Mizar article ではありませんから、この２つのエラーメッセージを理解する必要はありません。　では、Mizar がこれをどう取り扱うのかを見たので、この「**hello Mizar!**」を削除しましょう。　エラーメッセージは自分で削除する必要はありません。　次に **mizf** を駆動したときに、このプログラムが除去してくれます。

　エラー行は **::>** で始まるので、それとわかります。　Mizar はエラー番号だけを出力します。　決して、Mizar がなぜその行が正しくないと考えるかはプリントしません。　時には説明がないことでイライラすることもありますが、概して Mizar のエラーは、まったくわかりやすいもの（obvious）です。

Exercise 1.2.1　最小の Mizar article は前に書いたように２行です。　では MML の最小の article は何行でしょう？　最長の article は？　MML の article の平均的な行数は？

　おおざっぱにいって、1000 行以下の article は MML に投稿するには短すぎると考えられます。　しかし、いくつかの article は MML の書き直しの過程で短縮されるので 1000 行以下のものもあります。　現在 MML 中の 1000 行以下の article はいくつあるでしょうか？

Exercise 1.2.2　ローカルの **text** ディレクトリに MML の中の article の１つをコピーしてみてください。　Mizar checker（**mizf**）がエラーメッセージを出さずにこれを処理するのを確認してください。　article をちょっとだけ変更して、Mizar が改ざんした箇所を検出できるかどうか実験してください。

　スクリーン１面分の Mizar のものではないテキスト　― 例えば Pascal あるいは C のプログラム ―　を Mizar article の真ん中あたりに挿入してみてください。　どのくらいの数のエラーメッセージを貰いました？　Mizar はエラーから十分に回復できて、ファイルの残りの部分をチェックできましたか？

１．　３　　Step 3：the statement 「ステートメント」

数学の記述を Mizar 文に翻訳し始めるには、証明しようとする定理を Mizar の文法に従って書く必要があります。

Mizar の文法については２つの重要な点に注意しなければなりません：

- Mizar においてはスペルの寛容度（spelling variants）は全くありません。 Mizar は自然言語にとてもよく似た文法規則を持っていますが、形式化言語であり、言い回し（phrasing）選択の余地はありません。　例えば：

  and と **&** は別物です。
  not と **non** は別物です。
  such that と **st** は別物です。
  assume と **suppose** は別物です。
  NAT と **Nat** は別物で **natural** もまた別物です。

  ですから、注意深く Mizar 文法の正確なキーワードを使用する必要があります。　「似ている」ことは何の役にも立ちません。　この規則の唯一の例外は、**be** と **being** でどちらのスペルでも互換性があります。 **let X be set** そして **for X being set holds**… と書くほうがより自然であっても、**let X being set** や **for X be set** と書くことも許されます。

- Mizar では ‘function notation’（関数記述）と ‘operator notation’（演算子記述）の区別はありません。　ほとんどのプログラム言語では **f(x,y)** のような記述と **x+y** のような記述は文法的に区別されます。 Mizar にはこの区別はありません。 Mizar ではすべては operator（演算子）です。　Mizar で **f(x,y)** と書いたとすると、左の引数を持たず、右の引数を２つ持つ演算子シンボル（‘operator symbol’）**f** ということになります。

  同様に述語の名前（predicate names）と型の名前（type names）は希望する文字列を使って大丈夫です。　好みにおうじて文字と数字やその他のキャラクタを混ぜて使ってください。　例えばMizarでは述語を **¥/-distributive**と命名することも可能です。　それは単一のシンボルになります。　どのキャラクタがMizarのシンボルとして他のシンボルと一緒に使えるかが確実でなければ、MML abstractsのウェブページ（訳注：http://mizar.org/library）で調べることができます。　そこではシンボルにその定義部分を参照するためのハイパーリンクが張ってあります。

Mizar を書くのには terms（項）と formulas（式）の書き方を知らねばなりま

せん。　これにはトップダウン方式で取り組みます。　はじめに、述語論理式をどのようにMizarの形式にあてはめるかについての方法を書きます。　次にMizar言語の文法がどのようなものかを記します。

1.3.1 Formulas 「式」

以下に述語論理式を書くために必要なすべてものの表をのせます。

⊥	**contradiction**
¬φ	**not** φ
φ ∧ ψ	φ **&** ψ
φ ∨ ψ	φ **or** ψ
φ ⇒ ψ	φ **implies** ψ
φ ⇔ ψ	φ **iff** ψ
∃x. ψ	**ex x st** ψ
∀x. ψ	**for x holds** ψ
∀x.(ψ ⇒ ψ)	**for x st** ψ **holds** ψ

Mizarでは ⊤ を書く特別の方法はありません。　普通 not contradiction に、この記号 ⊤ を使います。　量化子（quantifier）と一緒に使用するときは **st** 、そうでないときは **such that** であることに注意してください。

さらに、**for x st** φ **holds** ψ は **for x holds**（φ **implies** ψ）のシンタックス・シュガーにすぎないことにも注意して下さい。　構文解析部（parser）が構文処理をした後では、システムはこの２つの式にいかなる差異も認めません。　このテーブルを使って、Mizarの論理式を書くことができます。　実際にMizar式を書くためには、もうひとつ知らなければならない事があります。 Mizarは型の概念を持つプログラム言語（a *typed* language）です。次のSection 1.3.4でMizarの型について詳細に議論しますが、型の問題は述語論理式にもあります。　式の中で使うすべての変数は型を持つ必要があります。変数に型を与えるには２つの方法があります：

- 式のなかで型付けを行うには **being** という型の属性（type attribution）を書いてください。　例えば、自然数 **m,n** が存在するという式を書くのに：

  ex m,n being Nat st …

  と書くことができます。

- **reserve** ステートメントで変数を型付けしてみてください。　これは変数そのものは導入せず、表記規約（訳注：変数の型の取り決め、notation convention）

を導入します。 **reserve** を：

```
reserve m,n for Nat;
ex m,n st …
```

と使っておけば式の中での型の記入は不要です。 このやり方による変数の型付けは、明示的に変数を型付けする方法よりも便利なので、ひろく一般的に使われています。

Exercise 1.3.1 次の Mizar の式を述語論理式による記述に変換してください：

φ **iff not not** φ

not (φ **&** ψ) **implies not** φ **or not** ψ

ex x st φ **implies** ψ

for x st for y st φ **holds** ψ **holds** χ 。

この練習問題を解くのにはMizarの論理演算子（logical operator）の優先順位（priority）についての知識が必要です。

ex/for < implies/iff < or < & < not

これは **not** φ **implies** ψ ではなく、**(not φ)implies ψ** と読み（なぜなら**not**の結合がより強いので）、**ex x st** φ **implies ψ** は **ex x st(φ implies ψ)** を意味する、と読まねばなりません。

Exercise 1.3.2 次の述語論理式をMizar文法の式に変換しなさい：

¬φ ⇔ ¬¬¬φ

¬（φ ∨ ψ）⇒（¬φ ∧ ¬ψ）

∃*x.*（φ ∧ ψ）

∃*x.*（（∃*y.*（φ ⇒ ψ））⇒ χ） 。

1.3.2 Relations 「関係」

さらに、原子式（atomic formulas）の書き方を知る必要があります。 Mizar でこれを書くには２つ方法があります：

- もし***R*** が述語論理のシンボル（predicate symbol）ならば、

x_1, x_2, ..., x_m **R** *x_{m+1}, ..., x_{m+n}*

m, n あるいはその両方とも0（zero）でも良いのですが、その場合は prefix あるいは postfix 記述となります。 postfix 記述の場合は、たとえば **x, y are_relative_prime** のように一つ以上の引数を持つことができるのに注意してください。 また述語の両側の括弧は許されません。

- もし *T* が型、あるいは型の形容詞部分（adjective part of a type）ならば：

  x* is *T

  と書きます。 例えば、**x is prime**と書くことができます。 type（型）については後の **Section 1.3.4** で議論しましょう。

  どういう関係（relation）が使えるのか、MMLをブラウズしてみてください。Mizar 勉強の初めのほうで最も頻回に使われるいくつかの relation を短いリストにしておきます。

=	**=**
≠	**<>**
<	**<**
≤	**<=**
∈	**in**
⊆	**c=**

Mizar の⊆記号の最初のキャラクタは小文字の c です。

1.3.3 Terms 「項」

Mizar の term（項）を書くのに、３つの方法があります：

- **f** が operator symbol（演算子シンボル）なら、Mizar では *functor*（関数記号）と言いますが、次のように書いてください：

  (x_1, x_2, ..., x_m) f (x_{m+1}, ...x_{m+n})

  繰り返しますが、**m** あるいは **n** は 0（zero）でも良く、その場合は prefix 記法あるいは postfix 記法になります。 例えば postfix 記法は **^2** が自乗の記号として使われます。 左右どちらの引数の括弧も引数が１つしかない場合には省略可能です。

- ***L*** と ***R*** をカッコのシンボル（bracket symbol）として、次のように書いて

ください。

L* x_1, x_2,…, x_n *R

この場合、引数の両側のカッコ‘(’と‘)’は、シンボルそれ自身がカッコなので、必要ありません。 この種の項（term）の例には順序対（ordered pair）**[x, y]** があります。 その場合 **n=2**、***L*** は‘**[**’に、***R*** は‘**]**’に相当します。

- どんな自然数も Mizar の項（term）になります。 自然数を書くときは、environ に以下の行を追加します。

```
requirements SUBST, NUMERALS, ARITHM;
```

そうしないとそれ（訳注：自然数）は自然数として振舞いません。

functor という関数記号のシンボル（function symbol）あるいは演算子のシンボル（operator symbol）は Mizar の術語（terminology）です。 圏論（category theory）では、functor の考えは意味を持ちません。 それは論理の中で function symbol を集合論の function object（これは対の集合のことです）を区別するのに使われます。

もう一度言いましょう、どの operator が使用可能かを知るには MML をブラウズする必要があります。 ここに、初めのうちに最もよく使われると思われる演算子（operator）の一部ですが、それの短いリストを挙げておきます。

$\emptyset$	**{}**
$\{x\}$	***{x}***
$\{x, y\}$	***{x,y}***
$X \cup Y$	***X ¥/ Y***
$X \cap Y$	***X /¥ Y***
$\mathbb{N}$	**NAT**
$\mathbb{Z}$	**INT**
$\mathbb{R}$	**REAL**
$x + y$	***x + y***
$x - y$	***x - y***
$-x$	***-x***
xy	***x*y***
x / y	***x/y***
x^2	***x^2***
x^y	***x to_power y***
$\sqrt{x}$	**sqrt *x***

$P(X)$	**bool *X***
(x, y)	**[*x* ,*y*]**
$X \times Y$	**[:*X*, *Y*:]**
Y^X	**Funcs(*X*,*Y*)**
$F(x)$	***f*.*x***
$\langle\rangle$	**<*>*D***
$\langle x \rangle$	**<* *x* *>**
$\langle x, y \rangle$	**<* *x*, *y* *>**
$p \cdot q$	***p*^*q***

数字の 2 は平方（square）の operator の一部です。　最後の 4 個の operator は集合 ***D*** の有限数列（finite sequence）を作るのに使われます。

・(dot) 演算は集合論内部 *(inside the set theory)* での関数の応用例 (application) です。　もし ***f*** が言語の中のシンボルで左引数を持たず右引数を 1 つ持つ functor のことであるなら次のように書けます：

```
f x
```

あるいは（いつでもカッコの中に term（項）を書けます）

```
f(x)
```

と書けます。　***f*** が集合論の function ならば　　— 対の集合ですが —

```
f.x
```

は ***f*** の像（image）になります、言い換えると **y such that [x, y] in *f*** となります。

Exercise 1.3.3　　次の Mizar 式を数学の記述に変換しなさい：

```
NAT c= REAL
1/0 = 0
sqrt -1 in REAL
sqrt(x^2) <> x
{x} /¥ {-x} = {x} /¥ {0}
[x, y] = {{x, y},{x}}
p = <*p.1,p.2*>  。
```

Exercise 1.3.4　　次の数学の式を Mizar 式に変換しなさい：

$\sqrt{xy} \leq \frac{x+y}{2}$

$(-1, \sqrt{2}) \in \mathbb{Z} \times \mathbb{R}$

$X \cap Y \subseteq X \cup Y$

$Y^X \in P(P(X \times Y))$

$\langle x, y \rangle = \langle x \rangle \cdot \langle y \rangle$

$p \cdot \langle\rangle = p$

$f(g(x)) \neq g(f(x))$ 。

1.3.4 Types: modes and attributes 「型：モードと属性」

もう一つ Mizar 式を書くために理解しなければならないのは Mizar の型です。 Mizar は ZF（Zermelo-Fränkel）スタイルの集合論を基礎としています ― だからMizar のオブジェクトは型無しです ―。 Mizar の term（項）は型付きです。♨

例えばある Mizar の型は：

non empty finite Subset of NAT

です。 この型は：

のように構文解析されます。 Mizar の型は *mode*（モード）のインスタンス（instance）とその前に置く形容詞 *(adjectives)* の一群（cluster）から構成されます。 形容詞 *(adjectives)* を持たない型（type）は *radix type*（基本型）と呼ばれます。 この場合モードは **Subset** で引数は **NAT**、 基本型（radix type）は **Subset of NAT** で２つの形容詞（adjectives）は **non empty** と **finite** です。 これを抽象的に（訳注：一般性を持たせて）書くと、Mizar type は：

$\alpha_1, \alpha_2, \ldots, \alpha_m$ ***M*** **of** $x_1, x_2, \ldots, x_n$

と書けます。 ここで $\alpha_1, \alpha_2, \ldots, \alpha_m$ は形容詞（adjectives）、***M*** はモードのシンボル、そして$x_1, x_2, \ldots, x_n$ は terms（項）です。 **of** というキーワードはモードの引数をモードと結合します。 ちょうど、term（項）のカッコと同じように。 引数の数 **n** はモードの定義の一部です。 例えば、**set** の場合 **n** はゼロです。 **set of** … と書くことはできません、引数をとる **set** のモード

はありませんから。

モードは独立の型ではありません。 どういうモードを使うことができるかを知るためには、やはり MML をブラウズしてください。 以下は初めのうちに最も良く使われるモードの短いリストです。

set
number
Element of *X*
Subset of *X*
Nat
Integer
Real
Ordinal
Relation
Relation of *X, Y*
Function
Function of *X, Y*
FinSequence of *X*

Relation と **Function** の２つのモードは引数をとらず、より限定された（specific）モード **Relation of X, Y**と**Function of X, Y** は２つの引数をとります。 それらは **Relation** や **Function** などのシンボルを共通に使いますが別の物です。

さらにモードは terms（項）に依存します。 だから、ある型からある型への関数（function）を表す型はありません。 **Function** というモードは **set** から **set** への関数(function)を表します。 例として関数空間(the function space) (X→Y) × X→Y は Mizar の型の **Function of [:Funcs（X, Y）:], Y** に相当します。

形容詞（adjectives）は基本型（radix type）をサブタイプ（subtype）に修飾します。 形容詞（adjective）は、属性 *(attribute)* か、あるいはキーワード **non** を使って作られるその否定か、どちらかです。 どういう属性 *(attribute)* を使うことができるかを見つけるには、やはり MML をブラウズしましょう。 いくつか属性 *(attribute)* のリストを載せます。

empty
even
odd
prime

natural
integer
real
finite
infinite
countable

皆さんは、多分 Mizar でどのように型（type）を使うか悩んでいると思います。はっきりさせるには３種類の Mizar の概念 ― **NAT**、**Nat**、そして**natural** ― を検討する必要があります。 いずれも、自然数の集合という同一の物を意味します。

meaning	*declaration*	*formula*
n ∈ ℕ		**n in NAT**
n：ℕ	**n be Nat**	**n is Nat**
n：ℕ	**n be natural number**	**n is natural**

NAT は term（項）、**Nat** は type（型）で、**natural** は adjective（形容詞）です。 **be/being** は宣言のなかでの型付けで、変数と型の間に入ります。**in** は集合論の∈関係で、２つの term（項）の間に入ります。 **is** は term と type の間、あるいは term と adjective の間に入ります。 Mizar では論理チェックで型の判定を利用できます。

例： 例としてある定理のステートメントを提示します。 この章で証明しようとするのは：

```
reserve a, b, c, m, n for Nat;

a^2 + b^2 = c^2 & a, b are_relative_prime & a is odd implies
  ex m, n st m <= n & a = n^2 - m^2 & b = 2*m*n & c = n^2 + m^2;
```

です。 さあ、これを **my_mizar.miz** ファイルの **begin** の行の次に追加してください。 さらに **a, b,** そして **c** を明示的に：

```
for a,b,c st
 a^2 + b^2 = c^2 & a,b are_relative_prime & a is odd holds
  ex m,n st m <= n & a = n^2 - m^2 & b = 2*m*n & c = n^2 + m^2;
```

と量化（quantification）することも考えられます（could have made）が、しかしこれは不要です。 Mizar が自動的に自由変数の量化（quantify）をして

くれます。　さてこのステートメントを詳細に解析してみましょう。　その式は：

φ1 & φ2 & φ3 implies ex m,n st φ4 & φ5 & φ6 & φ7;

という構造を持っています。　これに対応する述語論理式は：

$$(\varphi_1 \wedge \varphi_2 \wedge \varphi_3) \Rightarrow \exists m, n.\ (\varphi_4 \wedge \varphi_5 \wedge \varphi_6 \wedge \varphi_7)$$

です。　この中で、はじめの３つの原子式（atomic formula）は：

φ1 ≡ **a^2 + b^2 = c^2**
φ2 ≡ **a,b are_relative_prime**
φ3 ≡ **a is odd**

です。　これらは：

φ1 ≡ *t*1 **=** *t*2

φ2 ≡ *t*3, *t*4 **are relative prime**

φ3 ≡ *t*5 **is** *T*

という構造をもっています。　このなかで、**=** と **are_relative_prime** は述語論理のシンボル（predicate symbols）です。　*T* は 形容詞（adjective）**odd** です。　さて、ここで２種類の原子式を見ることになります： term の間の relation を２度、型付け（typing）を１度です。

φ1 の中の最初の term *t*1 は：

*t*1 ≡ **a^2 + b^2**

で：

*t*1 ≡ *u*1 + *u*2

*u*1 ≡ *v*1^2

*u*2 ≡ *v*2^2

*v*1 ≡ **a**

*v*2 ≡ **b**

という構造を持っています。　この中で、**+** と **^2** は関数記号（functor）のシ

ンボルです。

Exercise 1.3.5　以下のコンセプトの Mizar の型を見つけなさい：

素数でない奇数。

自然数の空の有限数列。

実数の不可算集合。

空集合の要素。

空集合の空でない部分集合。

最後の２つのコンセプトの問題点は何でしょう？　それらは Mizar で許されると思いますか？　許されるかどうかこのマニュアルを良く研究してください。

Exercise 1.3.6　下のステートメントを Mizar 式で書きなさい：

偶数の素数は２のみである。

もし素数がある積を割り切るならそれは因数の一つを割り切る。

最大の素数というのは存在しない。

４より大きい素数はいずれも２つの素数の和である。

これらのステートメントを最初は **reserve** を使って、次に式の中の型付けを使って書いてください。

１．４　Step 4：　getting the environment right 「正しい環境部を書く」

書いたステートメントを article に追加してください。　そしてチェックして下さい。　エラーメッセージ：

```
environ
 vocabularies MY_MIZAR;
begin
 reserve a,b,c,m,n for Nat;
::>                      *151
```

```
a^2 + b^2 = c^2 & a,b are_relative_prime & a is odd implies
::>,203
ex m,n st m <= n & a = n^2 - m^2 & b = 2*m*n & c = n^2 + m^2;

::> 151: Unknown mode format
::> 203: Unknown token, maybe the forbidden underscore
character used in an identifier
```

をもらうでしょう。 それはまだ MML から引用して使用している部分をインポートしていないからです。 ステートメントが悪いのではなく、環境部に不備があります(環境部はまだ空です)。 これを修正するには、*指令(directives)* を **environ** 部に加えることにより、必要とする部分をインポートすることが必要です。

Mizar article の環境部を正しく書くことはなかなか難しいのです。 実際は、古い article の環境部をコピーして使うことが多いのです。 しかし、それで動かないときには、そのやり方は役に立たないし、また実際ちょくちょく動かないことがあります。 そこで、やはり環境部がどう働くのかを理解し、環境部がらみのエラーが出たときは正しく対処しなければなりません。

Exercise 1.4.1 MML にすでにある環境部を使ってみてください。 エラーはなくなりましたか? もしなくなったのなら、さしあたり *Step 4* の残りを考えながら、その環境部を使っていてください。

1.4.1. **Vocabulary, notations, constructors**「ボキャブラリ、ノーテーションズ、コンストラクターズ」

この規則は全く単純です。 使うすべてのもの — predicate, functor, mode あるいは attribute — それらに該当する参照を:

- **vocabularies**
- **notations**
- **constructors**

の３つの指令文（directive）に適切に追加しなければなりません。 一般に **notations** と **constructors** の参照リストの内容はほとんど同じものです。実際このセクションのアルゴリズムに従って環境部を正しく働くようにすれば、その２つは同一になります。 それらの指令（directives）は:

- 語彙分析（Lexical analysis）。 Mizar articleの表象（tokens）は

vocabularies と呼ぶリストから読み込まれます。 Mizar の変数は一定の文法を持つ識別子（identifier）ですが、predicate（s）、functor（s）、type（s）、attribute（s）などはすべて、あらゆるキャラクタを含むシンボル *(symbols)* です。 シンボルをどの **vocabularies**（語彙）から使うかを指示する必要があります。

- 表現の構文解析（Parsing of expressions）。 表現を使う場合は、predicate(s)、 functor(s)、type(s)、attribute(s) が入っている article のリストが必要になります。 **notations** という指令は表現の文法（syntax）のためにあります。 **constructors** という指令は表現の意味（meaning）のためにあります。

（訳注：symbols、tokens、expressions、terms の差異を的確に訳出できません）

に関するものです。 以下にステートメントで使用するもの、その種類、それから必要な vocabularies と articles のリストを置きます。

symbol	*kind*	*vocabulary*	*article*
=	*pred*		**HIDDEN**
<=	*pred*	**HIDDEN**	**XXREAL_0**
+	*func*	**HIDDEN**	**XCMPLX_0**
*****	*func*	**HIDDEN**	**XCMPLX_0**
-	*func*	**ARYTM_1**	**XCMPLX_0**
Nat	*mode*	**HIDDEN**	**NAT_1**
are_relative_prime	*pred*	**ARYTM_3**	**INT_2**
^2	*func*	**SQUARE_1**	**SQUARE_1**
odd	*attr*	**MATRIX_1**	**ABIAN**

HIDDEN *vocabulary* と **HIDDEN** *article* 以外のすべてのリストを明示的に参照する必要があります。 *vocabularies* は **vocabularies** directive のなかに、articles は **notations** と **constructors** 指令の両方に入れる必要があります。 環境部は：

```
environ
 vocabularies MY_MIZAR, ARYTM_1, ARYTM_3, SQUARE_1, MATRIX_2;
 notations XXREAL_0, XCMPLX_0, NAT_1, INT_2, SQUARE_1, ABIAN;
 constructors XXREAL_0, XCMPLX_0,NAT_1, INT_2, ABIAN;
```

のようになっているはずです。 与えられたシンボルがどの vocabulary や、

どの article にあるかを探す方法を以下に示します：

- vocabulary を探すにはコマンド：

  findvoc -w *'symbol'*

  を使います。 例えばコマンド：

  findvoc -w '-'

  は：

  ```
  FindVoc, Mizar Ver. 7.0.01 (Win32/FPC)
  Copyright © 1990, 2003 Association of Mizar User
  vocabulary: ARYTM_10
  0- 32
  ```

  を返します。 **O** はこのシンボルが関数記号シンボル（functor symbol）であることを示し、**32** は優先度を表します。

- article を探すのには、Mizar ウェブサイト（訳注：http://mizar.org）の MML abstract へ行くのが一番簡単な方法で、まずどこかでそのシンボルを使用している場所を探し、それをクリックしてその定義に行くのが良いでしょう。

Exercise 1.4.2 **listvoc HIDDEN** とタイプして： **HIDDEN vocabulary** に何があるか調べなさい。

このボキャブラリの 25 のシンボルの各々について、それを使っている記数法（notation）を採用（introduces）している article を見つけなさい。

1.4.2 Redefinitions and clusters 「再定義とクラスタ」

しかし、事態はなかなか手ごわいのです。 環境部は依然正しく働いていないのが明らかになります！ 問題は型が正しくないことです。 新しい 環境部で article をチェックすると：

```
a^2 + b^2 = c^2 & a, b are_relative_prime & a is odd implies
::>  *103
ex m, n st m <= n & a = n^2 - m^2 & b = 2*m*n & c = n^2 + m^2;
::>          *102          *103        *103          *103
::> 102: Unknown predicate
::> 103: Unknown functor
```

とコメントをもらいます。　この種の型にまつわる問題を解決するのは Mizar article を書く上で最も困難な部分です。　（実際上この種のエラーはしばしば cluster が行方不明（missing）になっていることに起因します。　ですから Mizar の型にまつわるエラーで完全に途方に暮れた時は、まず ‘cluster’ のことを考えてください。）

実際、ここで見るエラーのほとんどは、いくつかの cluster が環境部で導入されなかったために起きます。　その結果、表現（expression）が正しい型を持っていないことになってしまいます。　例えば、最初のエラーは **a^2** と **b^2** が **Element of REAL** という型を持つのに、 **+** は引数として複素数型（**complex number**）という型を要求することで起きます。　この場合なぜそうなるかを詳細に示し、正しいクラスタ（cluster）を導入（importing）することでこの問題を解決しようと思います。

Mizar の表現（expression）は唯一つの型を持つわけではなく、型の集合（whole set of types）を保持します。　例えば正しい環境部では、**2** という数字は以下の複数の型を持ちます：

```
Nat
natural number
prime Nat
Integer
integer number
even Integer
Real
real number
Complex
complex number
Ordinal
ordinal number
set
finite set
non empty set
. . .
```

ある表現（expression）の型に影響を与えるには２つの方法があります。

- ある特定の表現（expression）に、より厳密（precise）な radix type（基本型）を与えるためには*再定義（redefinitions）*を使います。　functor はその引数の型に従って異なる型を持つことができるので、多数の redefinition

を持つことができます。　ここでは **PEPIN** からとった redefinition の例：

```
definition let n be Nat;
  redefine func n^2 -> Nat;
end;
```

を挙げます。　この再定義は、**^2** 演算子を使う項（terms）の型の変更をします。　この元の定義（definition）、（それもやはり redefinition です）は **SQUARE_1** にあり：

```
definition let x be complex number;
  func x^2 -> set equals x * x;
end;
```

となっています。　**SQUARE_1** にはすでに：

```
definition let x be Element of COMPLEX;
  redefine func x^2 -> Element of COMPLEX;
end;

definition let x be Element of REAL;
  redefine func x^2 -> Element of REAL;
end;
```

という **^2** の別の再定義が２つあります。　さて環境部に：

```
notations ... , SQUARE_1, PEPIN, ... ;
```

という指令があるとしましょう。　もし ***t*^2** という表現を書けば、この表現の型は ***t*** の型に依存します。 **^2** のすべての定義と再定義は **notations** 指令を介してインポートされた順番で考慮され（will be considered）ます。　この場合、最初は複素数（**complex number**）の定義が、つぎに **Element of COMPLEX** という再定義、それから **Element of REAL** の再定義、最後に（これは次の article、すなわち **PEPIN** 中にありますが）**Nat** という再定義があります。

さて、規則では、すべての引数の（訳注：すべての）型に適合（fits）した再定義のうち、*最後*の再定義が適用（applies）されます。
もし ***t*** が **Nat** 型ならば ***t*^2** も**Nat** 型です、一方 ***t*** は **Element of COMPLEX** や **Element of REAL** や **Nat** の型のうちのどれも持たないとすると、**set** 型を持つことになります。

この理由により **notations** 指令の中の article の順番が重要であることに注意してください！

- 表現の形容詞（adjectives）を生成するために、Mizar はさらに *cluster* と呼ばれるものを持っています。 この cluster による形容詞の追加過程をクラスタの駆り集め（*rounding up* of clusters）と呼びます。

ここに **FINSET_1** からのクラスタの３つの例があります。

```
registration
  cluster empty -> finite set;
end;
```

これは形容詞 **empty** を持つどの **set** も同時に形容詞 **finite** を持つことになることを意味します。

```
registration let B be finite set;
  cluster -> finite Subset of B;
end;
```

これは **B** が **finite set** 型を持つとき、**Subset of B** 型を持つどの表現も形容詞 **finite** を持つことになることを意味します。

```
registration let X,Y be finite set;
  cluster X ¥/ Y -> finite;
end;
```

これは **X** と **Y** が **finite set** 型を持つ時、どの **X ¥/ Y** の形（shape）の表現も、 形容詞 **finite** を持つことを意味します。 これらの例は、表現の型の集合に形容詞（adjective）を追加する２種類の cluster を示しています。最初の２つはこれ（形容詞の追加）を表現の*型（type）*に基づいて行い（これは‘型を駆り集める（“rounding up” a type）’といいます）、３つ目は表現の*形（shape）*に基づいて形容詞を追加します。

まとめると: 再定義(redefinitions)は基本型(radix type)を狭める(narrowing)ために、そして cluster は形容詞の集合を拡げる（extending）ためにあるということになります。（さらに型（typing）になんの作用もしない再定義もあり、― それらは型と違うものを再定義します ― そして、第３の種類（kind）の形容詞（adjective）の追加に関して何もしない cluster もあります。 これらを混同しないようにしてください）

ある種の型 *T* が ***t*** という表現の持つ型の集合の中にあるかどうかを、*t* をその型に変更することで（*t* **qua** *T*）いつでもテストできます。 もし *t* がその型の集合に *T* を含んでいなければエラーが起きます。 その場合は状況を変える cluster や redefinition を探し始めると良いかも（might start）しれません。

例 この例の最初のエラー：

```
a^2 + b^2 = c^2 & a, b are_relative_prime & a is odd implies
::> *103
ex m,n st m <= n & a = n^2 - m^2 & b = 2*m*n & c = n^2 + m^2;
::>          *102           *103         *103            *103
::> 102: Unknown predicate
::> 103: Unknown functor
```

では、**a^2 + b^2** の **+** は正しく型付けされていません。（エラーメッセージ ***103**の ***** はエラーの位置を示しています、この場合 **+** の真下にあります。）
ここで現在有効な **^2** の定義は **SQUARE_1** から：

```
definition let x be Element of REAL;
  redefine func x^2 -> Element of REAL;
end;
```

であり、そして **+** の定義は **XCMPLX_0** から：

```
definition let x, y be complex number;
  func x + y means ...
  ...
end;
```

となります。 そう、これはあなたが **Element of REAL** 型をとる **a^2** と **b^2** が、同時にまた **complex number** 型も持ってくれると好ましい、と思っていることを示しています。 その表現が、追加の形容詞 ― 形容詞 **complex** ― を持つことを望み、そのために cluster を必要としていることを意味します。
（もう一度： もっと形容詞が必要な時は、cluster で、より厳密な基本型（precise radix type）が必要なら、再定義（redefinition）です。）
さて、適切な cluster は **XCMPLX_0** 内にあることがわかりました：

```
registration
  cluster -> complex Element of REAL;
```

end; 。

registrations XCMPLX_0; を環境部に付け加えれば、最初のエラーは消えて：

```
a^2 + b^2 = c^2 & a, b are_relative_prime & a is odd implies
 ex m,n st m <= n & a = n^2 - m^2 & b = 2*m*n & c = n^2 + m^2;
::>          *102                          *103

::> 102: Unknown predicate
::> 103: Unknown functor
```

と、２つエラーが残ります。 今度のエラー（**<=** の下の ***102**）は前のエラーと類似です。 **<=** 述語は **real number** 型の引数を期待します、しかし **m** と **n** は **Nat** 型を保持しています。 MML を少し研究すれば、 **Nat** は **Element of omega** と同じだとわかるでしょう（**NAT_1** の **Nat**の定義と **NUMBERS** の同義語（synonym）を見てください）。

というわけで、以下の２つの cluster は必要なものを与えてくれます。まず **ARYTHM_3** から：

```
registration
 ...
 cluster -> natural Element of omega;
end;
```

そして **XREAL_0** から：

```
registration
 cluster natural -> real number;
 ...
end;
```

です。 これで cluster 指令は：

```
registrations XCMPLX_0, ARYTM_3, XREAL_0;
```

となります。 環境部にこの指令を加えると、残るエラーは１つだけになって：

```
a^2 + b^2 = c^2 & a,b are_relative_prime & a is odd implies
 ex m,n st m <= n & a = n^2 - m^2 & b = 2*m*n & c = n^2 + m^2;
::>                                    *103
```

```
::> 103: Unknown functor
```

となります。　このエラーは Mizar が **2** を **Nat** と考えないことにより起きます。　次の **requirement** 指令に関する議論の後で、どうやってこの最後のエラーから逃れるかを見ましょう。

1.4.3　The other directives　「その他の指令」

以下に環境部（**environ**）の８種類の指令とそれらのとる参照の種類（kind）のリスト：

vocabularies	*vocabulary*
notations	*article*
constructors	*article*
registrations	*article*
definitions	*article*
theorems	*article*
schemes	*article*
requirements	**BOOLE, SUBSET, NUMERALS, ARITHM, REAL**

を提示します。　さあ、最後の４種類の指令が必要となる時が来ました：

- **definitions** 指令とは、**definitions** の一部である定理（theorem）を使うことをできるようにする指令では*ありません*。　（それは theorem 指令の一部です。）　これは、証明しようとしている命題の中での述語の自動的な展開に関するものです。　この指令は有用ですが、重要ではありません。Mizar 学習の調子が出てくるまでは無視していて結構です。

- **theorems** 指令と**schemes** 指令は、使う定理（theorems）とスキーム（schemes）をその中に持つ article のリストです。　だから証明の中で theorem を参照するときはいつでも article がこの指令のリスト中にあるかどうかチェックしないといけません。　これらは容易に正しい理解ができる指令です。

- **requirements** 指令は Mizar に適切なもの（appropriate things）を自動的に知らせる指令です。　例えば数字（numerals）に **Nat** 型を与えるには

```
requirements SUBSET, NUMERALS;
```

とする必要があります。　これは article に最後まで残された型に関するエラーの解決法です。　**ARITHM** により Mizar はある種の数（numbers）についての基本的な式（equations）を自動的に使えるようになります。　それにより、例えば、Mizar は証明なしに **1+1 = 2** を認めて（accept）くれます。　これも容易に正しく理解できる指令です。　すべての必要な requirements を追加して、終わらせてしまいましょう。

さて、これで環境部は正しくなりました。　article のチェックをすると：

```
environ
vocabularies MY_MIZAR, ARYTM_1, ARYTM_3, SQUARE_1, MATRIX_2;
notations XXREAL_0, XCMPLX_0, NAT_1, INT_2, SQUARE_1, ABIAN;
constructors XXREAL_0, XCMPLX_0, NAT_1, INT_2, SQUARE_1,
ABIAN;
registrations XCMPLX_0, ARYTM_3, XREAL_0;
requirements SUBSET, NUMERALS;

begin
 reserve a, b, c, m, n for Nat;
a^2 + b^2 = c^2 & a, b are_relative_prime & a is odd implies
ex m, n st m <= n & a = n^2 - m^2 & b = 2*m*n & c = n^2 + m^2;
::>                                                          *4

::> 4: This inference is not accepted
```

のようになっているでしょう。　唯一の残ったエラーは ***4** です。　それは Mizar がステートメントを自分だけでは証明できなかったことを意味します。これは重要な到達点（mile stone）です。　残っているのがエラー ***4** だけの article は‘ほぼ完成’です。　あとはただ、もうすこし証明が必要なだけですから。

Exercise 1.4.3　次のそれぞれのステートメントに対し、***4** 以外のエラーがすべて無くなる環境を見つけなさい：

```
for X, Y being non empty set,
 f being Function of X, Y, g being Function of Y, X st
  f is one-to-one & g is one-to-one holds
   ex h being Function of X,Y st h is bijective
```

```
for p being FinSequence, D being set st
 for i being Nat st i in dom p holds p.i in D holds
 p is FinSequence of D;

for G being Group, H being Subgroup of G st
 G is finite holds ord G = ord H * index H;

for GX being TopSpace, A,C being Subset of GX st
 C is connected & C meets A & C ¥ A <> {}GX holds
 C meets Fr A;  。
```

Exercise 1.4.4　次の12種類の型：　（訳注：//??　9種類→12種類？）

```
Element of NAT
Element of INT
Element of REAL
Element of COMPLEX
Nat
Integer
Real
Complex
natural number
integer number
real number
complex number
```

を考えてみて下さい。　このリストからT_1とT_2の異なる型の８１対が：

for x being T_1 holds (x qua T_2) = x;

に対して許されるはずです。（しかしそれらのすべてが MML で証明可能というわけではありません）。　エラーメッセージ ***116**（無効な "**qua**"！）の数を最小にする環境部はどのようなものですか？　この環境の article から何が使われますか？

Exercise 1.4.5　Section 1.4.2 で示した cluster の種類（kind）（項に追加の形容詞を生成するやつ）の他に、Mizar には*存在性のクラスタ（existential clusters）*と呼ばれるものがあります。　これは **->** を持たない clusters です。

これは項に追加の形容詞を生成しません。　そのかわり、型への形容詞の追加が許可されていることが必要です。　この理由は Mizar の型は常に **non-empty** でないといけないからです。　そう、ある型の使用が許されるのには、何かが証明されねばなりません（訳注：So to be allowed to use a type something has to be proved.）。　存在性のクラスタ（existential clusters）がこの証明をします。

使うのが許される型の例として：

```
non empty finite Subset of NAT
```

という型がありますが、まず **GROUP_2** にある存在性のクラスタ（existential cluster）：

```
registration let X be non empty set;
  cluster finite non empty Subset of X;
end;
```

が必要です。 正しい存在性のクラスタ(existential clusters)が article にないと、未登録（non registered）の cluster を意味するエラー ***136** が出てしまいます。

MML の中で、次のリストのどの型が存在性のクラスタ(existential cluster)を持ちますか？　それを持つやつをどこで見つけましたか？　それを持たないやつに対しては、何が適切な存在性のクラスタ（existential cluster）になるでしょうか？

```
empty set
odd prime Nat
infinite Subset of REAL
non empty Relation of NAT, NAT
```

１．５　Step 5：maybe：definitions and lemmas 「多分：定義と補題」

ステップ５は飛ばして（skip）かまいません。　直ちに定理（theorem）の証明の仕事にかかってください。　しかしステップ１をちゃんとやった場合は、おそらく、これから必要になる補題がすでにいくつかあるのがおわかりでしょう。あるいは、とにかく定理を記述する前に、ある考え（notion）を明確に定義する必要がありそうです。　そう、それをこのステップでやりましょう。　article に

関係のある定義と補題をここで追加しましょう。　ちょうどステップ３のステートメントのように。　まだ、証明はしません。

1.5.1　Functors,　predicates　「関数記号、述語」

Mizar で functor を定義するのには２つの方法があります。

- （訳注：func と言う functor の）略語（abbreviation）として：

```
definition let x be complex number;
  func x^2 -> complex number equals x * x;
  coherence;
end;
```
。

一貫性の正当性条件（**coherence** correctness condition）というのは、x * x という表現が前に宣言された（claimed）ように複素数型（complex number）という型を実際に保持しているということです。　もし Mizar がそれ自身で明らかにできないのなら、あなたが証明しなければならないでしょう。

- 結果の性格付け（characterization）によって：

```
definition let a be real number;
  assume 0 <= a;
  func sqrt a -> real number means
    0 <= it & it^2 = a;
  existence;
  uniqueness;
end;
```
。

性格付けのステートメントでは、変数 **it** は functor の結果です。　くりかえしますが、**existence**（存在性）と **uniqueness**（一義性）は正当性の条件です。　それらは特性の定義（defining property）を満足する値が存在すること、その値が唯１つ（unique）であることを述べています。（証明の中でこの定義を満たしているという仮定を使うことが許されています。）

この場合、**existence** は：

```
ex x being real number st 0 <= x & x^2 = a
```

と言うステートメントです。　そして **uniqueness** はステートメント：

```
for x, x' being real number st
 0 <= x & x^2 = a & 0 <= x' & x'^2 = a holds x = x'
```

です。　以下に述語（predicate）を定義する方法の例：

```
● definition let m, n be Nat;

 pred m, n are_relative_prime means
   m hcf n = 1;
 end;
```

を載せます。　この述語の定義は正当性の条件を持ちません。（この例では、**hcf** は最大公約数です）。

1.5.2　Modes and attributes　「モードと属性」

Mizar ではモード（mode）を定義するのに２つの方法があります。　それらは：

● 略語として：

```
definition let X, Y be set;
  mode Function of X, Y is
    quasi_total Function-like Relation of X, Y;
end;
```

。

この定義で **quasi_total** と **Function-like** は属性（attributes）です。

● 要素（elements）の性格付け（characterization）によって：

```
  definition let X, Y be set;
    mode Relation of X, Y means
      it c= [:X,Y:];
    existence;
  end;
```

の２つです。　存在性の正当性条件（**existence** correctness condition）はモードが指定されて（inhabit）いることを述べています。　これは Mizar の論理（logic）がすべての Mizar の型は non-empty であることを要求するからで、この場合もあてはまります（This has to be the case）。　この場合、existence は：

```
  ex R st R c= [:X,Y:]
```

というステートメントです。　以下に属性（attribute）を定義する方法の例：

```
definition let i be number;
     attr i is even means
       ex j being Integer st i = 2*j;
  end;
```

を挙げます。

例　例題に必須の補題は、もし２つの数がお互いに（relative）素（prime）でその積が平方数（square）であるなら、それら自身はいずれも平方数である、というものです。　Mizar ではこれは：

```
m*n is square & m,n are_relative_prime implies
  m is square & n is square
```

というステートメントになります。 さて **square** という属性（attribute）はすでに MML に存在しますが、それは **PYTHTRIP** という article の中にあり、それはいま書こうとしているものと同じ内容です。　そこで、定義を書く練習として、それが MML に無いようなふりをさせて、article に追加してみましょう。　属性 **square**（これにより、**square number** という型を使用できます）を定義する方法は：

```
definition let n be number;
  attr n is square means
    ex m being Nat st n = m^2;
end;
```

となります。（ただ **Nat** 型の表現のためだけに **square** という attribute を定義したわけではなく、すべての Mizar の項（term）のために定義したことに注意して下さい。　その他の **not -1 is square** のようなステートメントは良く型付け（not be well typed）されていません。）

ほとんどの場合、定義の一部分は新しいシンボルの導入です。　この場合は **square** という属性のシンボルです。 くりかえしますが、これはすでに MML の vocabulary **PYTHTRIP** にありますが、それを使いたくないので、今書いている **MY_MIZAR** の **vocabulary** に書き加えましょう。　そこで：

```
  Vsquare
```

という一行を **my_mizar.voc** ファイルに加えます。

一番前の **V** は属性のシンボルを意味します。 **vocabularies** は functor の前には **O** を、predicate の前には **R** を、mode の前には **M** をattribute の前には **V** をつけます。 Mizar article のステートメントは article にローカルであるか、あるいは article の外部から可視であるか、のどちらかです。 後者の場合は **theorem** というキーワードが先行していなければなりません。 そこで：

```
theorem Th1:
 m*n is square & m, n are_relative_prime implies
  m is square & n is square;
::>                    *4,4

theorem Th2:
 a^2 + b^2 = c^2 & a, b are_relative_prime & a is odd implies
  ex m, n st m <= n & a = n^2 - m^2 & b = 2*m*n & c = n^2 + m^2;
::>                                                    *4
::> 4: This inference is not accepted
```

のようにステートメントの前に **theorem** を加えてください。（**theorem Th1** の下に２つ ***4** エラーがあるのに注意してください。 この理由は Mizar checker は **m** が平方数で（**m is square**）、 **n** も平方数（**n is square**）であるという２つの結論を、２つの異なる証明の要請（proof obligations）と見做したからです。

Exercise 1.5.1 略語による Mizar の定義は性格付け（characterization）による定義に対し二次的であることを示しなさい。

どうすれば、**func** の定義の **equals** 変種（equals variant）を **func** の定義の **means** 変種（means variant）でシミュレートできますか？ 同様に： どうすれば **mode** 定義の **is** 変種を **mode** 定義の **means** 変種でシミュレートできますか？

Exercise 1.5.2 Mizarの型はつねに **non-empty** であることを求められています。 しかし：

```
Element of {};
```

という型を書くことはできます。 なぜこれが問題にならないのか説明してください。

１．６　Step 6： proof　skeleton　「証明のスケルトン」

Mizar は Pascal プログラム言語に似たブロック構造化言語です。
begin/end ブロックの代わりに **proof/end** ブロックを持ちます。
procedure の代わりに theorems です。　それ以外は（訳注：２つの言語は）むしろ類似の言語と言ってよいでしょう。
　さて定理の証明を書いてみましょう。　さしあたりちょっとステップを書いて、それらの結合（link）はしないでおきましょう。　ステップの結合は「９つのやさしいステップ」のステップ７でやります
　Mizar の証明は **proof** キーワードで始め **end** キーワードで終了します。

重要！　もし、ステートメントの後に証明（proof）を書くつもりならば、ステートメントの後にセミコロンをつけてはいけません。　ですから：

statement ;
proof
　proof steps
end;

は間違いです！　それは：

statement
proof
　proof steps
end;

でなければなりません。　間違えやすいのですが、そこにあるべきでないセミコロンがあると ***4** エラーが出ます。

　本セクションでは次の種類の証明ステップを議論します。　もっとたくさんありますが、ここに示す：

step	*section*
compact	1.6.2
assume	1.6.1
thus	1.6.1
let	1.6.1
take	1.6.1
consider	1.6.2

per cases/suppose	1.6.2
set	1.6.3
reconsider	1.6.3
iterative equality	1.6.4

は良く使われるステップです。

1.6.1 Skeleton steps 「スケルトンステップ」

証明の段階では、証明されるべきステートメントを見失わないようにしなければなりません。 それは証明の‘ゴール’です。 それは*命題（thesis）*と呼ばれ、キーワード **thesis** で参照されます。 Mizar の証明はステップからなります。 ステップは与えられた文脈（context）の中で真であるとことを知っているステートメントを含みます。 ある種のステップは命題を次第に縮小します。 証明の最終部分では命題は ⊤ まで縮小されているはずです、さもないと「まだ証明すべき何かが残っています（**Something remains to be proved**）」を意味する、***70** エラーが出ます。 命題（thesis）を変更するステップは*スケルトンステップ（skeleton steps）*と呼ばれます。

４つの基本的なスケルトンステップは、**assume**、 **thus**、 **let** そして **take** です。 それらは：

thesis to be proved, before	*the step*	*thesis to be proved, after*
φ implies ψ	**assume φ**	**ψ**
φ & ψ	**thus φ**	**ψ**
for x being *T* holds ψ	**let x be *T***	**ψ**
ex x being *T* st φ	**take *t***	**ψ[x := *t*]**

のように命題の形（shape）に対応します。 **for** に関しても同様なのですが、変数が **reserve** ステートメントの中にあれば **let**（訳注：**let** ステップの変数）による型付けは省略可能（can be left out）です。 以下にこれらのスケルトンステップが非常に単純な証明中でどのように働くかの例：

```
for x, y st x = y holds y = x
proof
 let x, y;
 assume x = y;
 thus y = x;
end;
```

を示します。　証明の開始時に命題は **for x, y st x = y holds y = x** です。 **let** ステップの後で **x=y implies y=z** に縮小します。　**assume** ステップの後では、命題は **y=x** に縮小します。　**thus** ステップの後で、それは ⊤ にまで縮小され、証明は完成します。

（訳注：ピタゴラスの定理の）例の証明のスケルトンステップは：

```
theorem Th2:
  a^2 + b^2 = c^2 & a, b are_relative_prime & a is odd implies
    ex m, n st m <= n & a = n^2 - m^2 & b = 2*m*n & c = n^2 +
m^2
proof
  assume a^2 + b^2 = c^2;
  assume a, b are_relative_prime;
  assume a is odd;
  take m, n;
  thus m <= n;
  thus a = n^2 - m^2;
  thus b = 2*m*n;
  thus c = n^2 + m^2;
end;
```

のようになるでしょう。　この時点で ***m*** と ***n*** の場所に代入するものはなにもありません。　このためにもう１つの別の種類のステップが必要になります。

Exercise 1.6.1　なぜ証明の終了をいつも **thus thesis** のステップにできるのか説明しなさい。 これは QUOD ERAT DEMONSTRANDUM（訳注：Q.E.D.、証明終わり）の Mizar バージョンです。　MML の中に何回この構成が出てくるか数えてみてください（**hence thesis** も同じものなので、これも含めて数えなさい）。　MML の中でこの構成の一部でないような **thesis** が存在しますか？

Exercise 1.6.2　次の証明の Mizar ステートメントのためのスケルトンステップを書きなさい：

```
x = 0 & y = 0 implies x + y = 0 & x*y = 0

ex x st x in X implies for y holds y in X

for n st not ex m st n = 2*m holds ex m st n = 2*m + 1
```

(ex n st n in X) implies
ex n st n in X & for m st m < n holds not m in X 。

もし項（term）に対して適切な選択肢がないときは **take** ステップで新しい変数（fresh variable）を書いて下さい。

1.6.2 Compact statements and elimination 「コンパクトステートメントと消去」

もっとも簡潔なステップは *compact step* です。 これはそれだけで現在の文脈の中で真であるステートメントです。 もっとも良く使われる（common）Mizar の証明ステップです。 ほとんどのケース（most part）で証明のスケルトンはちょうど：

. . .
statement;
statement;
statement;
. . .

という形になります。 ここでリストの各々のステートメントは、先行するステートメントのいくつかの組み合わせ（combination）の帰結（consequence）です。

それから、**consider** と **per cases/suppose** という構文があります。 両方とも：

if you can prove this	*...*	*you can have this as a proof step*
ex x st φ		**consider x such that** φ**;**
φ_1 **or** φ_2 **or...or** φ_n		**per cases;**
		suppose φ_1**;**
		proof for the case φ_1
		end;
		suppose φ_2**;**
		proof for the case φ_2
		end;
		...
		suppose φ_n**;**
		proof for the case φ_n
		end;

というステートメントと関係します。　自然な推論（natural deduction）の術語（terminology）を使うと、これらのステップは存在証明（existential）と論理和（//??disjuction→disjunction）の消去（elimination）に対応しています。 **consider** ステップは証明の中で、ちょうど **let** ステップがするように、参照することができる変数を導入します。

さて、ここで、**take** ステップで項 **m** と **n** を導入するのに例題の証明の最後に **take** ステップを：

```
...
((c - a)/2)*((c + a)/2) = (b/2)^2;
((c - a)/2)*((c + a)/2) is square;
(c - a)/2, (c + a)/2 are_relative_prime;
(c - a)/2 is square & (c + a)/2 is square;
consider m such that (c - a)/2 = m^2;
consider n such that (c + a)/2 = n^2;
take m,n;
...
```

という具合に付け加えることができます。　最初の４つのステップはコンパクトステートメントです。　この２つの **consider** ステートメントは **square** の定義の一部である存在性のステートメント（existential statement）を使います。

1.6.3　Macros and casts　「マクロとキャスト」

証明中ではしばしばある特定の表現が何回も現れます。　そういう表現に名前を付けることが **set** コマンドを使って：

```
set h = b/2;
set m2 = (c - a)/2;
set n2 = (c + a)/2;
```

のように可能になります。　**set** コマンドは証明部（proof）の中の局所的定数の定義です。　それはマクロ（*macro* ）と非常によく似た振舞いをします。

Mizar article を書くときには、しばしばその表現が本来持っていない型を使いたいときがあります。　**reconsider** コマンドでその表現の型を変えることができます。　これは **set** に似ていますが、新しい変数の型（type of the new variable）を与えます。

この例では、短縮（abbreviated）された **h**、**m2** と **n2** は **Nat** という型を

持つ変数である必要があります。　これは前のパラグラフの **set** の行を：

```
reconsider h = b/2 as Nat;
reconsider m2 = (c − a)/2 as Nat;
reconsider n2 = (c + a)/2 as Nat;
```

に変えることで達成できます。

くりかえしますと、新しい変数はマクロのように振舞いますが、こんどは別の型を持っています。 そう、**reconsider** はある項(term)の型をキャスト(cast)する方法なのです。

1.6.4 Iterative equalities 「反復等号」

数学の計算では、しばしば等式（equality）の連鎖があります。 Mizar もまたこの特徴をもちます。 計算式：

$$\left(\frac{c-a}{2}\right)\left(\frac{c+a}{2}\right) = \frac{(c-a)(c+a)}{4} = \frac{c^2-a^2}{4} = \frac{b^2}{4} = \left(\frac{b}{2}\right)^2$$

は、Mizar では：

```
((c - a)/2) * ((c + a)/2) = (c - a)*(c + a)/4
.= ((c^2 - a^2)/4
.= b^2/4
.= (b/2)^2;
```

と書きます。 この **.=** の連鎖は*反復等号 (iterative equality)*と呼ばれます。連鎖の最初の等号は **.=** ではなく **=** であることに注意してください。

Exercise 1.6.3 この章の **Th2** の証明のステップを正しく直しなさい。**my_mizar.miz** ファイルで正しい順序に並べ替えてください。 （**set** 行ではなく）**reconsider** 行を使って、短縮された表現はすべてその短縮形（abbreviation）で置き換えてください。 反復等号も同様に書いてください。

b は偶数で **c** は奇数であることを述べる２つのコンパクトステートメントを **reconsider** 行の前に付け加えてください。 それらは **reconsider** 行をジャスティファイするのに必要です。 **m2**、**n2** と **a, c** に関する２つのコンパクトステートメントを **reconsider** 行の後に付け加えなさい。

さて、ファイルを **mizf** でチェックしましょう。 必要なら環境部をアップデートして ***4** エラーだけが出るようにしましょう。 いくつ ***4** エラーが出ましたか？

１．７　Step 7:　completing the proof　　「証明部の仕上げ」

このセクションでは証明にジャスティフィケーションのステップを加えます。これで、証明は完成です。

1.7.1　Getting rid of the *4 errors　　「＊４　エラーの退治」

Mizar ではステップのジャスティフィケーションには３つの方法があります：

- セミコロン；をその後におく方法：

  statement ;

これは空のジャスティフィケーション。　それは Mizar に：*「自分で明らかにしなさい」* と告げます。

- そのあとに **by** と先行ステートメントにつけたラベルの参照リストを書く方法：

  statement　**by** *reference, … , reference*;

実は、一つ前に書いたセミコロンを置く方法は、**by** の後の参照リストが空である特別の場合に相当します。

- サブプルーフをその後におく方法：

  statement　**proof**　*steps in the subproof*　**end ;**

  サブプルーフは Mizar の証明法にブロック構造化言語の外観を与えるものです。　もしサブプルーフで **thesis** キーワードを使用しないならば、Mizar はそのサブプルーフのスケルトンステップからステートメントを再構築（reconstruct）することができます。　ですから、ステートメントを除去できたら良いと思うことがあれば、その時は **now** を使えばそれが可能になります。

  now　*steps in the subproof*　**end;**

  これはサブプルーフを持つステートメントと正確に同等（the same as）ですが、（訳注：**now** があれば）ステートメントを書く必要はありません。

の３つです。　ここに **by** がどう使われるかを示す小さな例を書きます：

```
(x in X implies x in Y) & x in X implies x in Y
proof
  assume
A1: x in X implies x in Y;
  assume
A2: x in X;
  thus x in Y by A1,A2;
end;
```

もちろんこれは非常に簡単なので Mizar が自分で：

```
  (x in X implies x in Y) & x in X implies x in Y ;
```

とやってしまいます。 **by** によるジャスティフィケーションの後の参照リストでは、しばしばその直前のステートメントへの参照が起きます。 これはステップの前に **'then'** というキーワードを置けば達成できます。 **then** を使えば、しばしばラベル・ステートメントを回避できます。 そこで：

A1: *statement;*
A2: *statement;*
statement **by A1,A2:**

を

A1: *statement;*
statement;
then *statement* **by A1;**

で置き換えることができます。 ある人たちは：

A1: *statement;*
statement; **then**
statement **by A1;**

というように、 **then** を参照する行の前の行で、参照されるステートメントの同じ行の後のほうに書くのを好みます。 これは好みの問題です。 別の人達は **then** の位置を **then** の後のステートメントがラベルを持つかどうかで選んでいます。

thus ステップを **then** を使ってジャスティファイしようとしても、**then thus** と書くことは許されていません。 **hence** と書かねばなりません。 Mizar では、２つのキーワードがある特定の組み合わせで一緒になるときは、

別のものに代えないといけません。　ここに関係のある２つの式：

```
then + thus = hence
thus + now = hereby
```

を書いておきます。　**hence** キーワードは２つのことを意味します。

１．　これは前のステップを引き継ぎ（follows）ます。
２．　これは証明されることになっている式の一部です。

一般的に共通に使われる Mizar の慣用的表現は **hence thesis** です。　いろいろな Mizar article でお目にかかっていますね。

1.7.2　Properties and requirements　「特性とリクワィアメンツ」

さて、MML 内の関係のある定理の検索（hunt）が必要になります。　これは：

- MML は巨大です。
- MML は体系的に並べられているのではなく、経時的に配列されています。それは、収集された順に本棚に本を並べた中世の図書館に似ているのです。
- Mizar は巧妙（smart）です。　使うことができる定理は、探している定理とよく似ているとは言えないかも知れないのです。

などの３つの理由でなかなか困難です。　定理がいかに予期されない場所に存在しうるか、という例として Mizar library にある２種類の整除性（dividability）に関する基本的な定理：

```
m divides n iff m divides (n qua Integer);
```

の場合を挙げます。　これは SCPINVR という article のなかにあり、それはある小さなコンピュータプログラムのループ不変式（loop invariant）に関するものです（SCPはSCMPDSプログラムを意味し、SCMPDS は small computer model with push-down stack を意味します）。

MML で定理を探す一番良い方法は、すべての **.abs** ファイルに **grep** コマンドを適用することです。　MML は巨大ですが、この方法が実用的に使える程度の規模です。　例えば、この定理を **SCPINVAR** article の中で探すのには以下のコマンド：

```
% grep 'divides .* qua' $MIZFILES/abstr/*.abs
/usr/local/lib/mizar/abstr/scpinvar.abs: m divides n iff m
```

```
divides (n qua Integer);
%
```

が適切です。　定理を探す上で、もう一つの問題は Mizar があまりにも巧妙（smart）すぎることです。

　もし **+, *** や **<=** の定義を見れば、それらが **commutativity、reflexivity、onnectedness、synonym、 antonym** などのキーワードを含むことを発見するでょう：

```
definition let x,y be complex number;
 func x + y means
:: XCMPLX_0:def 4
  ...
  commutativity;
 func x * y means
:: XCMPLX_0:def 5
...
  commutativity;
end;

definition let x,y be ext-real number;
 pred x <= y means
:: XXREAL_0:def 5
...
   reflexivity;
  connectedness;
end;

notation let x,y be real number;
  synonym y >= x for x <= y;
  antonym y < x for x <= y;
  antonym x > y for x <= y;
end;  。
```

これらのキーワードは２つのことを意味します：

- もし同義語（**synonym**）あるいは反義語（**antonym**）を使おうとすれば、Mizar は内部的に実際の名前（real name）を使います。　そこで：

a < b と書けば、Mizar は内部的に：

not b <= a

の省略型だと考えるでしょう。　もし：

a >= b

と書けば Mizar は内部的には：

b <= a

を意味していると考えるでしょう。

- Mizar はステップをジャスティファイするときはいつでも **commutativity**、 **reflexivity**、 **connectedness** などの特性（properties）を知っています。　そこで：

  … **x*y** …

  とステートメントを書くと、Mizar はそのステートメントは正確に：

  … **y*x** …

  と同じであるとして振舞います。　さらに Mizar はジャスティフィケーションをしようとした時に、この形のすべての含意（implication）、すなわち：

  x = y ⇒ x ≤ y

  と：

  x ≤ y ∨ y ≤ x

  を使用します。
  これらの特性は、< 関係について：

  ¬ (x < x)

  と：

  x < y ⇒ x ≤ y

  を意味します。
  例えば：

  a < b

を証明済みであれば：

```
a <> b
```

だけが必要なときでもこれ（**a < b**）を参照して証明することができます。

これらはすべて非常に素晴らしいのですが、なかなか微妙なこと（subtlety）になる、３つの例を見てみましょう：

- 次のステップ：

```
a >= 0;
then c - a <= c by ...;
```

をジャスティファイしたいとしましょう。　必要な定理は：

```
theorem :: REAL_2:173
  (a<0 implies a+b<b & b-a>b) & (a+b<b or b-a>b implies a<0);
```

だとわかります。　これは：

```
c-a > c implies a < 0
```

を与えます。　しかし **<** と **>** の反義語（antonym）の定義からこれは実は：

```
not c-a <= c implies not 0 <= a
```

を意味し、それは：

```
0 <= a implies c-a <= c
```

と等価（equivalent）です。　さて必要なのはどれでしょう？

- 今、次の式：

```
(c + a + c - a)/2 = (c + c + a - a)/2
```

を証明する必要があるとしましょう。　**+** と **-** 演算子は同一の優先度を持ち左結合性（left associative）なので、これは：

```
(((c + a) + c) - a)/2 = (((c + c) + a) - a)/2
```

と読むべきです。　**+** の交換律（commutativity）からこれは実は：

```
((c + (c + a)) - a)/2 = (((c + c) + a) - a)/2
```

と同じです。　そこで：

c + (c + a) = (c + c) + a を、それは **XCMPLX_1** article のなかの定理：

```
theorem :: XCMPLX_1:1
  a + (b + c) = (a + b) + c;
```

は + の結合律なのですが、を示す必要があります。　もとの式が結合律の例（instance）とはまったく似ていないのに注意してください。

- **<=** に対する推移律（transitive law）を考えてみましょう：

  ```
  theorem :: AXIOMS:22
    x <= y & y <= z implies x <= z;
  ```
  。

  この定理があるのは良いことなのですが、どこに **<** に類似の法則（law）があるのでしょう：

  ```
  x < y & y < z implies x < z;
  ```
  。

  これもまた **AXIOMS:22** であることが明らかになります！　なぜこれがその場合にあたるのかは、それがより強い定理：

  ```
  x <= y & y < z implies x < z;
  ```

  の帰結であることに注意してください、そして **<** の反義語の定義からこれは：

  ```
  x <= y & not z <= y implies not z <= x;
  ```

  と同じで、それはまた：

  ```
  z <= x & x <= y implies z <= y;
  ```

  と等価（equivalent）で、それは確かに **AXIOMS:22** なのです。

（ここで例として使用した算術的演算（arithmetical operation）のための‘算術的補題（arithmetical lemmas）’は全て、**AXIOMS, REAL_1, REAL_2** と **XCMPLX** の article に入っています。　規則（rule）はもし探している補題が複素数でも有効（valid）ならば（一般的にはそれが等号（equality）や否定（not）や **<=**、**<** を含むことを意味します。）、それは **XCMPLX_1** にあります。　運悪く、それが不等号（inequality）のこと言っているのならば、他の３つの article をすべて探さなければなりません。　その場合は探しているものがそこにある

はずだと言う事を教えてくれる簡単なルールはありません。）

時々定理を探すべきでない状況があります。　もし **requirement ARITHM** があれば、Mizar は助けなしに自然数に関するたくさんの事実を知ることになります。　それらの（自然数に関する）事実がライブラリにはもはや定理を持っていないので、**requirement** を使うことを知らなければ、時には長いこと探し続けることになるでしょう。

例えば、**requirement ARITHM** を使えば：

```
x + 0 = x;
```

と：

```
0 * x = 0;
```

と：

```
1 * x = x;
```

と：

```
0 < 1;
```

と：

```
0 <> 1;
```

はいずれもジャスティフィケーションを必要としません。　いま：

```
a >= 1;
then a > 0 by ...;
```

をジャスティファイしようとしているとしましょう。　もし、すでに **0 < 1** を得ているなら、これは（ちょうど今見たように） **AXIOMS:22** の実例（インスタンス）でしょう。　しかし **requirement ARITHM** はただで（free）それを与えてくれます！　だからこのステップを **AXIOMS:22** だけでジャスティファイすることができます。

Exercise 1.7.1　MML は巨大だと断言（claim）しました。　MML の **.miz** article が何行のソースコードを持つか数えて見て下さい。　１人の有能な Mizar の作者が一日に何行書くか想像してみて下さい、また MML には何人-年分のソースがあるのか推定してみてください。

1.7.3 Automation in Mizar 「Mizar における自動化」

Mizar では他の proof checker では可能な、証明のための努力の一部を自動的に行うための*戦術（tactics）*を記述することはできません。 すべての Mizar の自動処理機能は Mizar system の中に開発者によって組み込まれています。

Mizar は４種類の自動処理を行います。

Semantic correlates（意味の相関形） Mizar は内部に式（formulas）を∧、¬、∀だけを使った、ある種の論理積正規形（conjunctive normal form）として格納します。

これは、自動的にある等価（equivalent）な式を識別することを意味します。これらの同値類（equivalence classes）、あるいは式（formulas）は *semantic correlates* と呼ばれます。 *semantic correlates* により式のスケルトン・ステップは、期待する（you might expect）以上にさらにパワフルなのです。 例えば：

```
φ
proof
 assume not φ;
 ...
 thus contradiction;
end;
```

を証明できます。 そして：

```
φ or ψ
proof
 assume not φ;
 ...
 thus ψ;
end;
```

も証明できます。 （Mizar は φ & ψ と ψ & φ を同一視しません。 証明の中に各命題に対する **thus** ステップを結合子（conjuncts）が現れる順番と同じ順に置かなければなりません。）

propertiesと**requirements** propertiesとrequirementsは前のセクションで議論しました。

clusters クラスタは表現に自動的に形容詞(adjectives)を生成します。 しばしば定理はクラスタに書き直す（rephrased）ことができ、その結果それらは

自動化されます。

例えば次の定理：

```
m is odd square & n is odd square implies m + n is non square;
```

を考えてください。　これは平方数がけっして２つの奇数の平方数の和にはならないと言っています。　（この理由は奇数の平方数はいつも４を法として１で、その２つの数の和は２を法として４ですが、偶数の平方数はいつも４を法として０だからです。）

この定理はクラスタとして書き直す（rephrase）ことができます。

```
definition let m,n be odd square Nat;
 cluster m + n -> non square;
end;
```

一度このクラスタを証明しておけば、Mizar は適当なときにこの定理を自動的に適用し、証明中でこれを（明示的に）参照しなくてもよいのです。

by を使ったジャスティフィケーション　**by** というジャスティファィヤはある種の弱い一次の証明子（prover）です。　それは **by** の前にあるステートメントを推論（deduce）しようとして **by** の後にあるステートメントを参照します。

by によるジャスティフィケーションはかなり‘自然の理由付けステップ（natural reasoning step）’に似た印象があります。　ある人間の書いた証明に関する研究は、ヒトは **by** がするのよりも、いくらか小さいステップを選ぶ傾向にあることがわかります。

by のとる方法は：

論理積正規形で証明しなければならない含意（implication）を置きます。それから個別の結合子（conjunct）を別々に処理しようとします。　これが、１つのジャスティフケーションの失敗でしばしば複数の ***4** エラーをもらう理由です。Mizar はジャスティファイできない結合子の１つにつき１つの ***4** エラーを出します。

- それから推論（inference）を証明しようとします。　先行する項のうちの１つだけが、全称量化子（universal quantifier）のインスタンス作成（instantiate）が可能です。　それゆえ：

  ```
  A1: for x holds x in X;
  ```

```
A2: for x holds not x in X;
contradiction by A1,A2;
```

はうまくいきません、なぜなら **for** を **A1** と **A2** の両方のインスタンスを作成（instantiate）しなければならないからです。 これは：

```
A1: for x holds x in X;
A2: for x holds not x in X;
   a in X by A1;
then contradiction by A2;
```

のように２つに分ける必要があります。

- **by** による証明子（prover）はそれが知っているすべての等式（equality）の一致による終了(congruence closure)を行うでしょう。 また等しい項(term)の型の情報を結合を行うでしょう。
- 存在性の量化子（existential quantifier）をもつ結論（conclusion）は全称量化子を持つ前項（antecedent）と等価（equivalent）です。 それゆえ **by** はそのインスタンス化されたもの（instantiations）から存在性のステートメント（existential statement）を引き出す（derive）ことができます。

1.7.4 Unfolding of definitions 「定義の展開 」

ここに Mizar が定義を展開（unfold）するときのテーブルがあります。

func / equals	−
func / means	−
pred / means	◇
mode / is	+
mode / means	◇
attr / means	◇
set	+

＋ と記された項（item）はマクロのように振る舞います。 例えば **Nat** という定義は、**Nat** を **Element of NAT** を書くように厳密に書くと、以下のようになります：

```
definition
  mode Nat is Element of NAT;
end;
```

そこでは、**Element** の **mode** についての規則（theorem）は自動的に **Nat** に適用されます。

テーブルのなかで ー と記されたアイテムは決して展開（expand）されません。 得られるのはすべて概念（notion）についての定義の規則（definitional theorem）です。 その定義を使うときはその規則（theorem）を参照することになります。

テーブルのなかで ◇ と記されたアイテムは、definitions 指令を使うなら展開されます。 展開（expansion）は命題（thesis）の中だけで発生します。 もし命題の形（shape of the thesis）と合致しないようなスケルトン・ステップが実行されようとすると展開が起きます。

例えば **c=** の定義は：

```
definition let X, Y;
  pred X c= Y means
:: TARSKI:def 3
    x in X implies x in Y;
  reflexivity;
end;
```

です。 もし **definitions** に **TARSKI** があれば、集合の包含（set inclusion）を：

```
X c= Y
proof
  let x be set;
  assume x in X;
  ...
  thus x in Y;
end;
```

と証明することができます。 同様に、**XBOOLE_0** で equality の定義：

```
definition let X,Y;
redefine pred X = Y means
:: X_BOOLE_0:def 10
  X c= Y & Y c= X;
end;
```

を考えてみましょう。 もし **definitions** に **XBOOLE_0** を加えるなら、以

下の証明が許されます：

```
X = Y
proof
  ...
  thus X c= Y;
  ...
  thus Y c= X;
end;
```

あるいは：

```
X = Y
proof
  hereby
    let x be set;
    assume x in X;
    ...
    thus x in Y;
  end;
  let x be set;
  assume x in Y;
  ...
  thus x in X;
end;
```

でさえも証明が許されます。

Exercise 1.7.2　次の３つのステートメントの Mizar proof を：

```
reserve X,Y,Z,x for set;

for X holds X c= X;

for X,Y st X c= Y & Y c= X holds X = Y;

for X,Y,Z st X c= Y & Y c= Z holds X c= Z;
```

。

MML から以下の２つの定理：

```
theorem :: TARSKI:2
```

```
  for X,Y holds
    X = Y iff for x holds x in X iff x in Y;
theorem :: TARSKI:def 3
  for X,Y holds
    X c= Y iff for x st x in X implies x in Y;
```

だけを引用して使用し、書き上げなさい。（もし MML の中を調べて、**TARSKI** article のなかにそれが確かに存在しないということが明らかになるようなら、教訓的（didactic）目的でそれと似たものを提示します。） 証明のなかで取るべきアプローチは、**X c= Z** のようなステートメントを証明するため、まず始めに：

```
...
A1: for x holds x in X implies x in Z;
proof
 ...
end;
...
```

を証明するべきです。 それから **X c= Z** を、**TARSKI:def 3** と一緒にこの **for** ステートメントを参照して：

```
...
X c= Z by A1, TARSKI def:3;
...
```

という具合にジャスティファイしなさい。 まず、その中での主張（argumentation）ができる限り明瞭な証明を書き、二番目にできる限り短い証明も書いてください。

Exercise 1.7.3 “*酒飲みの原理*“の証明をする次のスケルトンを採用してください。 これは、酒をのむ人達の部屋にいる人の集合を **X** とすると、各部屋には、*ある人が酒を飲むならその部屋の人全員が酒を飲むという（such that）人がいる*というので、”*酒飲みの原理*“ とよばれます。 これは逆説的に響きますが、(なぜ一人の人がその部屋の人すべてに酒を飲ませる権限を持つのか？) もし本当にこれが言っていること理解したなら、決して逆説的ではありません。

```
reserve X, x, y for set;
ex x st x in X implies for y holds y in X
proof
```

```
  per cases;
  suppose ex x st not x in X;
    consider x such that not x in X;
    take x;
    assume x in X;
    contradiction;
    let y;
    thus y in X;
  end;
  suppose not ex x st not x in X;
    for x holds x in X;
    assume x in X;
    thus thesis;
  end;
end;
```

ステートメントにラベルを付けて、すべてのステップについてジャスティフィケーションしなさい。　同じステートメントで別の証明も実験してみてください。　もし構成的論理（constructive logic）を知っているなら：　どれが非構成的ステップであるかを示した、構成的論理ステートメントの Mizar proof を書きなさい

Exercise 1.7.4　次のステートメントの証明を書きなさい。

```
reserve X, Y, x, y for set;
for X, Y, y holds
 ((ex x st x in X implies y in Y) &
  (ex x st y in Y implies x in X))
iff (ex x st x in X iff y in Y);
```

per casesを必要としない証明を考えることができますか？

Exercise 1.7.5　**Th2** の証明を完成しなさい。　必要な限りのクラスタと補題を追加しなさい。　例えば：

```
Lm1: m is odd square & n is odd square implies
      m + n is non square;
Lm2: n is even iff n / 2 is Nat;
Lm3: m,n are_relative_prime iff
```

```
  for p being prime Nat holds not (p divides m & p divides n);
Lm4: m^2 = n^2 iff m = n;
```

を使うのが良いかも知れませんが、他の適合する（訳注：//?? finc→fit）補題を使うのを選んでもＯＫです。　それらを定理の前に書き込んでください（良ければ後で証明してみて下さい）。　もし補題が他と較べて有用でないと思うならば、前に定理を置く必要はありません。

さあ、すべての ***4** エラーが消え去るまでコンパクトステートメントとジャスティフィケーションを **Th2** の証明に加えてください。

１．８　Step 8：　cleaning the proof　「証明のクリーニング」

さて、article が完成したので、面白いことがはじまります！　Mizar は、article の改良を助ける数種類のユーティリティを持っています。　これらのユーティリティは article のなかの不必要な部分を指摘してくれます。

それらのユーティリティは、エラーメッセージをファイルの中に書き込むことはしません。　そのユーティリティー・プログラムの名前をプログラム **revf** の引数として与える必要があります。

relprem　　このプログラムは **by** ジャスティフィケーション中の不必要な参照　― 不必要な **then** の出現（occurrence）も同様ですが ―　を指摘してくれます。　指摘された参照は安全に article から削除できます。

このプログラムはまた article の実際のエラーも指摘します。　ある人たちは **mizf** の代わりに常に **relprem** を使うのを好みます。

relinfer　　このプログラムは証明の中の不必要なステップを指摘します。それはショート・サーキットできるステートメントへの参照も指摘します。　そのような参照は除去できるし、その参照されるほうのステップを置き換えることができます。

relinfer の例として次の証明の部分を考えてみましょう。

```
A1: m2*n2 = h^2 by ... ;
A2: m2*n2 is square by A1;
A3: m2,n2 are_relative_prime by ... ;
A4: m2 is square & n2 is square by A2, A3, Th1;
 consider m such that m2 = m^2 by A4;
...
```

もし： **revf relinfer my_mizar** を走らせると、***604** という無関係の参照を意味するコメントを貰います。

```
A1: m2 * n2 = h^2 by ... ;
A2: m2 * n2 is square by A1;
A3: m2,n2 are_relative_prime by ... ;
A4: m2 is square & n2 is square by A2, A3, Th1;
::>                                    *604
 consider m such that m2 = m^2 by A4;
…
```

これはステップ **A4** の中の **A2** への参照を **A2** 自身のジャスティフィケーションで置き換えることを意味します。　この場合は **A1** です。　その結果：

```
A1: m2 * n2 = h^2 by ... ;
A3: m2,n2 are_relative_prime by ... ;
A4: m2 is square & n2 is square by A1,A3,Th1;
consider m such that m2 = m^2 by A4;
...
```

のように短くできます。　どうか、この **relinfer** プログラムが危険であることに注意してください！　なぜなら、証明が破壊されるだけではなく、それらが醜くなるかも（become ugly）知れないからです。　**relinfer** は人間が見たいと思うステップをカットするのを奨めるのです。

reliters（**re-liters**ではなく**rel-iters**と発音してください）　このプログラムはスキップできる反復等号（訳注：//??interative→iterative）のステップを指摘します。　もしそういうステップを削除したならばその参照先をその次のステップにする必要があります。

reliters の例として：

```
m2 * n2 = (c - a)*(c + a)/(2 * 2) by XCMPLX_1:77
 .= (c - a)*(c + a)/4
 .= (c^2 - a^2)/4 by SQUARE_1:67
 .= b^2/4 by A1,XCMPLX_1:26
 .= b^2/(2 * 2)
 .= b^2/2^2 by SQUARE_1:def 3
 .= h^2 by SQUARE_1:69;
```

の反復等号を考えてください。　**revf reliters my_mizar** を走らせると：

```
m2*n2 = (c - a)*(c + a)/(2*2) by XCMPLX_1:77
 .= (c - a)*(c + a)/4
::>                  *746
 .= (c^2 - a^2)/4 by SQUARE_1:67
 .= b^2 / 4 by A1,XCMPLX_1:26
::>          *746
 .= b^2 / (2*2)
 .= b^2 / 2^2 by SQUARE_1:def 3
 .= h^2 by SQUARE_1:69;
```

となります。 そこで ***746** のついた項(term)を反復等号から削除できます。それからジャスティフィケーションのなかの参照先を、一連の続きのなかの次の項（term）に変更せねばなりません。 それで反復等号は：

```
m2*n2 = (c - a)*(c + a)/(2*2) by XCMPLX_1:77
 .= (c^2 - a^2)/4 by SQUARE_1:67
 .= b^2 / (2*2) by A1,XCMPLX_1:26
 .= b^2 / 2^2 by SQUARE_1:def 3
 .= h^2 by SQUARE_1:69;
```

のように短くなります。

trivdemo このプログラムはあまりにシンプルなので、単一の **by** ジャスティフィケーションで置き換えることができるサブプルーフを指摘します。そういう場合がとても頻回に明らかになるので驚かされます。

chklab このプログラムは参照されていないすべてのラベルを指摘します。 指摘されたラベルは安全に article から除去できます。

inacc このプログラムは参照されていないすべての証明の部分を指摘します。 指摘されたものは安全に article から除去できます。

irrvoc このプログラムは使われていないすべての **vocabularies**を指摘します。 指摘されたものは安全に article から除去できます。

irrths このプログラムは **theorem** 指令の中で証明の参照のために使われていない article を指摘します。 指摘されたものは安全に article から除去できます。

article のクリーニングは面白いです！ それはまるでなにかを完成した後で、磨き上げて美しくする感じがあります。

最適化を終了した後でもう一回 **mizf** でチェックして、間違えて新しいエラーを持ち込んでいないのを確認するのを忘れないでください。

Exercise 1.8.1　Exercise 1.7.3 で作った証明に、このセクションで述べたプログラムを適用してみて下さい。　どのプログラムが証明を変更するようにアドバイスしてくれましたか？

Exercise 1.8.2　Mizar proof のステートメントをグラフの点と考え、ステートメントへの参照を辺と対応させて考えてみてください。　これらのグラフの変形という見方をしたときに、**relprem** と **relinfer** の最適化というのはどういうことをしているのでしょう。　絵を描いてみてください。

１．　９　Step 9：submitting to the library 「ライブラリへ投稿」

さて、article は完成しておりクリーンです。　MML へ投稿することを考えるべきです。

MML のために満たすべき２つの規則があります：

- まだ MML には無い数学である。
  MML の中の同じコンセプトの別の形式化であってはなりません。　MML にはすでに多くの重複した仕事があります。
- それは重要で意義深い。
  おおざっぱに言って article は少なくとも1000行は必要です。　もし短いと思うなら、十分な長さを持つまで拡張してください。

MML に article を投稿する理由はたくさんあります：

- 他の人たちがあなたの仕事を使えるし、その上にまた仕事を築きます。
- 後の Mizar article ではあなたの定義があなたの研究対象のスタンダードになります。
- Mizar システムが変わったとき、あなたの article は Mizar グループの人たちによって適合するように維持されるでしょう。
- 自動的に作成されたあなたの article の $\TeX$ 版が *Formalized Mathematics* というジャーナルに掲載されます。

MML に article を投稿する方法の詳細については Mizar project のウェブサ

イト(訳注:http://mizar.org)で見つけて該当するリンクをクリックしてください。 Article を MML に投稿するにあたっては、Mizar グループの人に彼らが良いと思うように article を変更する権利を与える書類に署名しなければなりません。

もし article が投稿にはむいていないけれども、それを次の article で使用したければ、ローカル・ライブラリに格納しておくことができます。 現在この種のローカル・ライブラリは、それが非能率になる前の程度、すなわち数個の article しかサポートしません。 MML は最適化(optimize)されているので、ライブラリファイルはあまり大きくならないのです。 ローカル・ライブラリを自分で最適化することは、そのためのツールが配布されていないので不可能です。 だからできるなら article を MML に投稿することは本当に良いことなのです。

もし自分でローカル・ライブラリを作るなら、既にある **dict** ディレクトリの次 (next) に **prel** という三番目のディレクトリが必要です。 このディレクトリにファイルをいれるのにはコマンド:

```
miz2prel text¥my_mizar
```

を **text** と **prel** ディレクトリの外で走らせます(複数のファイルが **prel** ディレクトリに入ります)。 **miz2prel** を走らせた後では、それ以降の article で以前の article を使うことができます。

Exercise 1.9.1 **MML には何人ぐらいの著者がいますか? あなたもそのなかの一人になってはいかがでしょう?**

第２章　Some advanced features

「いくつかの先進的特徴」

さて Mizar article をどのように書くか分りました、まだ知らない Mizar のちょっと進んだ特徴を見てみましょう。

これらの特徴を示すのに、Grzegorz Bancerek の小さな証明 **quantal1.miz** の article を使いましょう。 この article は *quantales*（訳注：クオンテイル；量子力学から発生した束の変種？）の数学的概念（notion）についてのものです。 quantales が何であるかを知ることは重要ではありません。ただ、Mizar のいくつかの特徴を示すためだけにこの証明を使います。

```
reserve
Q for left-distributive right-distributive complete
Lattice-like
     (non empty QuantaleStr),
 a, b, c, d for Element of Q;
theorem Th5:
 for Q for X,Y being set holds "¥/"(X,Q) [*] "¥/"(Y,Q) =
"¥/"({a[*]
b: a in X & b in Y}, Q)
  proof let Q; let X,Y be set;
 deffunc F(Element of Q) = $1[*]"¥/"(Y,Q);
 deffunc G(Element of Q) = "¥/"({$1[*]b: b in Y}, Q);
 defpred P[set] means $1 in X;
 deffunc H(Element of Q,Element of Q) = $1[*]$2;
A1: for a holds F(a) = G(a) by Def5;
  {F(c): P[c]} = {G(a): P[a]} from FRAENKEL:sch 5(A1);
hence
 "¥/"(X,Q) [*] "¥/"(Y,Q) =
 "¥/"({"¥/"({H(a,b) where b is Element of Q: b in Y}, Q)
   where a is Element of Q: a in X}, Q) by Def6 .=
 "¥/"({H(c,d) where c is Element of Q,
       d is Element of Q: c in X & d in Y}, Q)
   from LUBFraenkelDistr;
end;
```

この証明はいままでの章の例よりも、より数学の抽象的な性質（kind）に関するものです。　Mizar は抽象数学の形式化に特に適しているのです。

２．１　Set comprehension： the Fränkel operator 「集合の包含：フレンケル・オペレータ」

quantal1.miz の定理 **Th5** の証明は表現：

```
{ H(a,b) where b is Element of Q: b in Y }
```

を含みます。　これは：

```
{H(a, b)|b ∈ Y }
```

に相当（correspond）します。　ここで、**a** は固定のパラメターで、**b** は **b** ∈ **Y** を満たす **Q** の要素の範囲が許されます。

Mizar ではこのスタイルの包含は *Fränkel operator* と呼ばれます。　それは集合論の「置換公理」に相当し、最初にアドルフ・フレンケル（ZF の F です）が提唱しました。

そのフレンケル・オペレータの一般形は：

{ *term* **where** *declaration of the variables* : *formula* }　です。

証明中の他のいくつかのフレンケル・オペレータの項で見ることができるように、もし項の中のすべての変数が **reserve** ステートメントに現れるなら（訳注：書くなら）、変数の宣言は暗黙的にすんでしまいます。

フレンケル・オペレータを書くことが許されるには、関係する変数の型（type）が **Element of** *A* を持つ Mizar の型に*拡大（widen）*されている必要があります。　そうした時だけ、Mizar は定義された **set** が本当に **set** であることを知ることができます。　そうでない場合には、正当な（proper）クラスを：

```
{ X where X is set: not X in X }
```

のように書くこともできそうです（could:（訳注：subjunctive））。　しかし Mizar では、この表現は違法（illegal）です、なぜなら **set** は **Element of** の形（form）に拡大されていないからです。

Exercise 2.1.1　素数の集合は無限集合であるというステートメントを Mizar 文法にのっとって書き、証明しなさい。

２．２　Beyond first order ： schemes 「１階述語論理を越えて：スキーム」

quantal1.miz の証明の定理 **Th5** は **from** ジャスティフィケーションを使っています。　Mizar は一階述語論理を基礎にしています。　しかし１階述語論理は ZF スタイルの集合論を扱うのにはちょっと弱いのです。　１階述語論理に関するかぎり、集合論は無限にたくさんの公理を必要とします。　それが ZF 集合論が *axiom scheme*、置換公理を含む理由です。

どれくらい **scheme** と **from** が **theorem** と **by** と類似点を持つかを：

order	*item*	*used*
first order	**theorem**	**by**
higher order	**scheme**	**from**

というテーブルで示します。　スキームは高階論理ですが、それは非常に弱い意味においてです。　Mizar のスキームは 2 階ではなく 1.001 階ぐらいだと言われています。

はじめて証明に現れる **from** を研究しましょう。　それはつぎの等式：

```
{F(c): P[c]} = {G(a): P[a]} from FRAENKEL:sch 5(A1);
```

をジャスティファイします。　ジャスティフィケーションはステートメント：

```
A1: for a holds F(a) = G(a);
```

を参照します。　ここで使われたスキームは：

```
scheme :: FRAENKEL:sch 5
 FraenkelF' { B() -> non empty set,
  F(set) -> set, G(set) -> set, P[set] } :
  { F(v1) where v1 is Element of B() : P[v1] }
  = { G(v2) where v2 is Element of B() : P[v2] }
provided
 for v being Element of B() holds F(v) = G(v);
```

です。　スキームが証明したステートメントにはアンダーラインが引いてあります。　それは明らかに例題でジャスティファイされたものと同じステートメントです。　一般的なスキームの定義の構造は：

```
scheme label{ parameters}:
  statement
provided
  statements
proof
...
end;
```

となります。 **FRAENKEL:sch 5** スキームの場合、パラメターは **B, F, G** と **P** です。 最初の３つは関数（function）で、それらは丸カッコが続いて書かれています。 そして最後のものは述語（predicate）で角カッコを使って書かれています。

スキームを使うには、**deffunc** と **defpred** を使って、スキームのパラメターに適合したマクロ *(macro)* を書かねばなりません。 マクロの定義中ではマクロの引数は **$1,$2**… のように書かれます。 今回のような特別な場合では定義されたマクロは：

```
deffunc F(Element of Q) = $1[*]"¥/"(Y,Q);
deffunc G(Element of Q) = "¥/"({$1[*]b: b in Y}, Q);
defpred P[set] means $1 in X;
```

となります。（一見して，ここの **B ()** はマクロである必要はありません。 そこで Mizar は、この特別な例においては、**B ()** は **carrier of Q** でインスタンスを作成（instantiate）されなければならないことを自分で理解しています。）

スキームの他の例として、最も共通に使われるスキーム、つまり **NAT1:sch 1** を見てみましょう。 これは自然数についての帰納法（induction）を使えるようにします。 このスキームは：

```
scheme :: NAT_1:sch 1
  Ind { P[Nat] } :
    for k being Nat holds P[k]
  provided
    P[0]
and
 for k being Nat st P[k] holds P[k + 1];
```

となります。 さあ、このスキームを使っていくつか自然数に関するステートメントの証明をどのようにするのかお見せしましょう。 例として、自然数の

非負性：

```
for n being Nat holds n >= 0;
```

を考えてみましょう。　この帰納のスキームを使って明瞭な証明を得るため、つぎのマクロ：

```
defpred P[Nat] means $1 >= 0;
```

を定義しないといけません。　それから証明の構造は：

```
   defpred P[Nat] means $1 >= 0;
A1: P[0] by ...;
A2: for k being Nat st P[k] holds P[k + 1]
   proof
     let k be Nat;
     assume k >= 0;
     ...
     thus k + 1 >= 0 by ...;
   end;
   for n being Nat holds P[n] from NAT_1:sch 1(A1,A2);
```

となるでしょう。　ジャスティファイされるステートメントとスキームの引数の中に、マクロがなければなりません（さもないと Mizar はどうやってスキーム中のステートメントに合わせるのかを知らないままでしょう）、そこで **A1** の後に **0>= 0** と書けないのですが、どこでもすきなところで **P[K]** あるいは **k>= 0** と書くことはできます。　なぜなら、マクロ **P[K]** はただちにその定義に展開されるからです。

Exercise 2.2.1　Mizar のスキームのような自然の推論（deduction）の規則を書き、証明しなさい。　例えば、推論規則の三段論法（implication elimination（'modus ponens'））は：

```
scheme implication_elimination { P[], Q[] } : Q[]
provided
A1: P[] implies Q[] and
A2: P[]
  by A1,A2;
```

となります。

Exercise 2.2.2　Mizar の集合論の公理集は **TARSKI** という名前の article にあります。　それはタルスキー・グロタンディック（Tarski-Grothendieck）の理論と呼ばれる公理系を含むからです。　そして、それは ZF 集合論に巨大基数（濃度）存在の公理を加えたものです。　**TARSKI** article は一つのスキーム：

```
scheme :: TARSKI:sch 1
  Fraenkel { A()-> set, P[set, set] }:
    ex X st for x holds x in X iff ex y st y in A() & P[y,x]
provided
  for x,y,z st P[x,y] & P[x,z] holds y = z;
```

を持ちます。　今までのセクションのフレンケル・オペレータを使ってこのスキームを証明することは可能でしょうか（そして、MML の他のスキームを使用しないで）？

逆に、**TARSKI:sch 1** スキームを使って Mizar の証明からフレンケル・オペレータの使用を取り除くことができるでしょうか？

２．　３　Structures　「構造体」

quantal1.miz の定理 **Th5** の証明は：

```
left-distributive right-distributive complete Lattice-like
  (non empty QuantaleStr)
```

という面白い型（exciting type）を使います。　これらの型 **left-distributive**、 **right-distributive**、 **complete**、 **Lattice-like** と **empty** は以前と同じただの形容詞（attributes）です、しかしモード **QuantaleStr** は新しいものです。

QuantaleStr というモードは*構造体 (structure)* です。　構造体（structures）は Mizar の（訳注：Pascal で言う）**records** ですが、それは construction を使うときに選択可能な *fields*：

the *field*　**of**　*structure*

を保持します。　その例は次の表現：

```
the carrier of Q
```

です。　ここで、structure は **Q** で field の名前は **carrier** です。　たくさ

んの Mizar の構造体が **carrier** という名前の field を持ちます。

QuantaleStr 構造体の定義は：

```
struct(LattStr, HGrStr) QuantaleStr
  (# carrier -> set,
   L_join, L_meet, mult -> BinOp of the carrier #);
```

です。 見てのとおり４つのフィールド： **carrier**, **L_join**, **L_meet** と **mult** を持ちます。

LattStr と **HGrStr** のモードは構造体の*先祖型 (ancestors)* と呼ばれます。それらは定義された構造体のフィールドのサブセットを持ちます。
QuantaleStr という型を持つ項（term）は自動的に **LattStr** と **HGrStr** という型も持ちます。 **QuantaleStr** の先祖型（ancestor）の定義は：

```
struct(/¥-SemiLattStr, ¥/-SemiLattStr) LattStr
  (# carrier -> set, L_join, L_meet -> BinOp of the carrier #);
```

と：

```
struct(1-sorted) HGrStr (# carrier -> set,
                           mult -> BinOp of the carrier #);
```

です。 構造体は強力な概念です。 それは Mizar に抽象的な風合い（flavor）をもたらします。

Exercise 2.3.1 **VECTSP_1** の **Field** というモードの定義を見つけて下さい。その構造体は何の基礎の上に築かれているのでしょう？ 構造体の先祖型（ancestor）を再帰的にトレースしてみなさい。 それらの先祖型（ancestor）の間の拡長関係を図示しなさい。

Exercise 2.3.2 MML で最も基本的な（basic）構造体は、**STRUCT_0** の：

```
definition
  struct 1-sorted (# carrier -> set #);
end;
```

です。 **1-sorted** 型のオブジェクトはその **carrier** により一義的に（uniquely）定義されますか？ どうしてでしょう。 あるいは、そうでないならなぜでしょう。

Exercise 2.3.3　**1-sorted** 型のオブジェクト：

```
1-sorted (# A #)
```

を次のように構築することができます。　**set** 型のオブジェクトを **1-sorted** の中のその自然な対応（natural counterpart）に写像（map）する、あるいは逆写像する functor、**in** と **out** をこの記数法を使って定義して下さい。
Mizar で **out** は **in** の左逆関数（left inverse）であることを証明しなさい。

Exercise 2.3.4　ZF 構造体は集合でしょうか？　具体的には：　もし **Q** が構造体ならば、**x in Q** という **x** はありえますか？　もしだめなら、なぜでしょう？　もしそうなら以下を証明できますか？

```
ex Q being 1-sorted, x being set st x in Q
```

もしある構造体が要素（elements）を持てば：　構造体はその要素（elements）により一義的（uniquely）に決定されますか？

Exercise 2.3.5　ゲーデルの定理（Gödel's theorems）により、われわれは Mizar の集合論の不完全性（incomplete）を知っています。　すなわち、φ の肯定を証明するいかなる Mizar の証明も、φ の否定を証明するいかなる Mizar の証明も存在しない、自由変数を持たない Mizar formula φ が存在するに違いないということです。　Mizar で証明も否定もできない非ゲーデル的（non-Gödelian）　φ を想像することができますか？

索引　Index

S

T

U

V

X

い

か

き

く

け

こ

さ

し

す

せ

そ

た

ち

て

と

は

ひ

訳者あとがき

2007 年 4 月から信州大学工学部情報工学科、IT 大学院修士課程で師玉康成先生のクラスで数学、特に形式化数学の勉強を始めました。 Mizar を使うことで、数学の勉強のスタイルが変わる予感があります。 Mizar の入門書を探していて Freek Wiedijk 氏の著作を見つけました。 もともと個人の勉強で始めたものですが、メールを送ったところ、翻訳を Mizar people のグループで share しても良いとのお返事をいただいたので、とり急ぎ日本語化しました。 翻訳では Zton 氏に貴重なアドバイスいただきました。 ここに謝意を表します。 現時点では Mizar Version 7.8.05、MML Version 4.87.985 ですが、Mizar は in progress なので新機能や変更点にご注意ください（原著 mizman.pdf は 2006/3/7 に ESP Ghostscript で PDF 化されています）。 以下は Dr. Freek Wiedijk からのメール（全文）です。

2007.9 l'Hospitalier

Dear Kouichi Tamiya,

>On the day around my 60th birthday, I made up my mind to
>spend the rest of my life in learning Mathematics.

I don't know whether formalizing mathematics is the best way
to learn mathematics... （But it sounds like a nice plan :-））

>[...] and I am thinking of translation into Japanese by
>myself, for my own use. As you know, Mizar people form a
>small internal group in Japan, and if you are not reluctant,
>I am planning to share the translated book of your 'Writing
>a Mizar article in nine easy steps' with my colleagues.

That's fine with me. However, it would be nice if your
translation clearly states that I don't know Japanese,
so that I am not able to judge whether the translation
is correct. But do whatever you like with the text,
I don't mind.

Freek