Mizar in a Nutshell

（簡約 Mizar*）

ADAM GRABOWSKI
Institute of Mathematics, University of Białystok

and

ARTUR KORNIŁOWICZ and ADAM NAUMOWICZ
Institute of Informatics, University of Białystok

この論文は Mizar システムを使いはじめるのに役立つ Mizar の基礎的な術語についての実用的な参照マニュアルになることをめざしています。

日本語　3.1a 校正　　translation by l'Hospitalier　(2010.1-2010.3)

＊ in a nutshell : 【英】口語、「簡単に言うと」「要するに」「手短にいうと」。　O'Reilly出版に "Linux in a Nutshell" という本を含むシリーズがあります。

Journal of Formalized Reasoning Vol. 3, No. 2, 2010 Pages 153-245.

信州大学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興味で日本語に訳出しました。　文責はすべて l'Hospitalier にあります。　原著者には日本語への翻訳について連絡を取っていません。　個人的なMizar学習の目的以外の使用はお控えください。　使用者の便宜のために末尾に原著の p204-242、Appendix A, Appendix B, Acknowledgement, References のコピーを結合してあります（ページ番号が不連続です）。

皆様の勉強がすすみますように。

2011 年　早春　　l'Hospitalier

目　　次

1. 序文 (INTRODUCTION) ----------3

2. 言語 (LANGUAGE) ----------5
2.1 公式 (Formulas) ----------7
2.2 証明 (Proofs) ----------8
2.2.1 *証明の骨格 (Proof skeletons)* ----------9
2.2.2 ジャスティフィケーション (justification) ----------12
2.2.3 *補助的な証明の要素 (Auxiliary proof element)* ----------13
2.3 概念の定義 (Defining notions) ----------18
2.3.1 *述語 (Predicate)* ----------21
2.3.2 *属性 (Attributes)* ----------23
2.3.3 モード (Modes) ----------24
2.3.4 *ファンクタ (Functors)* ----------26
2.3.5 *構造体 (Structures)* ----------28
2.3.6 *同義語と反意語 (Synonyms and antonyms)* ----------30
2.4 再定義 (Redefinitions) ----------33
2.5 特性 (Properties) ----------36
2.5.1 *射影性 (Projectivity)* ----------37
2.5.2 *退縮性 (Involutiveness)* ----------37
2.5.3 *冪等性 (Idempotence)* ----------38
2.5.4 *可換性 (Commutativity)* ----------38
2.5.5 *反射性 (Reflexivity)* ----------38
2.5.6 *非反射性 (Irreflexivity)* ----------39
2.5.7 *対称性 (Symmetry)* ----------39
2.5.8 *非対称性 (Asymmetry)* ----------39
2.5.9 連結性 (Connectedness) ----------40
2.6 登録 (Registrations) ----------40
2.7 項の識別 (Terms identification) ----------43
2.8 定義,再定義と登録のまとめ(Summary of definitions, redefinitions and registrations)--45

3. システム (SYSTEM) ----------46
3.1 スキャナ-トークナイザ (Scanner-tokenizer) ----------46
3.2 パーサ (Parser) ----------47
3.3 アナライザ (Analyzer) ----------47
3.4 リーゾナ (Reasoner) ----------48

3.5 チェッカー（Checker）----------48
3.5.1 *スキマタイザ（Schematizer）*----------49

4. ソフトウェア（SOFTWARE）----------53
4.1 インストレーション（Installation）----------54
4.1.1 *Unixライクの OS達（Unix-like OS's）*----------54
4.1.2 *マイクロソフト ウィンドゥズ（Microsoft Windows）*----------56
4.2 Mizarアーティクルの準備 （Preparing a Mizar article）----------57
4.3 ボキャブラリ（Vocabularies）----------57
4.4 アコモデータと環境部宣言 （Accommodator and environment declaration）----------58
4.5 補助ユーティリティ（Auxiliary utilities）----------59
4.6 エンハンサ（Enhancers）----------60

5. Mizar数学ライブラリ（MIZAR MATHEMATICAL LIBRARY）----------63
5.1 公理（AXIOMATICS）----------63
5.1.1 *ファイル* HIDDEN（*file* HIDDEN）----------63
5.1.2 *ファイル* TARSK（*file* TARSKI）----------64
5.2 内容（Contents）----------66
5.3 アーティクルの投稿（Submission of articles）----------67
5.4 形式化数学 （Formalized Mathematics）----------69

6. Mizarについてのそれ以上の情報（MORE INFORMATION ON MIZAR）----------70

Appendix A. SKELETONS

Appendix B. THE SYNTAX OF THE MIZAR LANGUAGE

ACKNOWLEDGMENT

References

1．序文（INTRODUCTION）

Mizarとは Andrzej Trybulec により厳密に形式化された定義と証明を書くためにデザインされた形式記述言語の名前で、この言語で書かれた証明をチェックするコンピュータ・プログラムの名前でもあります。 Mizar project はこれらの（言語とプログラムの）両方の側面での開発を網羅（encompass）します。 便利な言語とソフトウェアの効率的な実装の組み合わせは、Mizarを豊富なアプリケーションをもつ証明援助手段として、また強力な教育ツールとして成功させています。 現在、Mizar community の努力の主な対象は、新しい article での使用、あるいは参照が可能な証明済みの定理、およびそれと相互関係にある定義の包括的なレポジトリ（貯蔵庫）である Mizar Mathematical Library (MML) を構築することです。

この論文は段階的に記述された Mizarチュートリアルではありません、むしろこの言語の表現力と検証システムについて書かれたリファレンス・マニュアルとでもいうべきでしょう。 主な目的は読者が基礎的なMizarの術語と仲間内の専門的な言葉の使い方（jargon）をよく知るのを助けることです。 読者はこの論文を勉強した後では MML で読むことができるすべてのMizarテキストを理解できるはずですし、自分の個人的な実験のために自分の証明を検証し自分用の証明を書くことでMizarを使いはじめられるようになっているはずです。

2．言語（LANGUAGE）

Mizar言語は数学論文で使われる言葉にできる限り近づくようにすると同時に自動的な検証ができるようにもデザインされています。 ２つのゴールは、非形式化数学でもっともよくあらわれる英語の単語（words）と句（phrases）の集合を選ぶことで実現されています。実際、Mizarは現実の文法（grammar）よりも数学の意味論的な慣用的表現（vernacular）にさらに近づくことを意図しています。 それゆえMizarの記法（syntax）は自然言語と比較して大幅に単純化されており、ある場合には英単語でなくその省略型が使われますが英語のスタイルとの差はあまりありません。

具体的には（In particular）言語は式の作成のため、そして定理のスキームを作成するため２階述語論理の自由変数を提供する手段として、１階述語論理の連結子と量化子の標準セットを含みます（定理のなかで無限を扱うファミリー、例えば数学的帰納法については Section 3.5.1を見てください）。 Mizarの統語構造（syntactic construct）の残りの部分は証明を書くこと、新しい数学的オブジェクトを定義することに使用されます。 この言語で書かれたテキストは通常、***アーティクル（article）***という名前でよばれます。

あらゆるアーティクルは80字より短い平文（ひらぶん、plain ASCII text）の ASCIIテキストで、キーワードenviron と begin ではじまります。 begin の後にはテキスト本体が続きます。 環境部は （訳注 : environ と begin の）２つのキーワードの間にはさまれていて、すでに存在する MML のアーティクルのどの概念をアーティクル本体が必要とするかをシ

ステムに知らせる８つのタイプの指令のうちの１つ以上（Section 4.4 を見てください）を指定し、そしてユーザが本体で定義したいと考える、新しいと考えられるエントリーのシンボルを含む外部ファイルの名前を指定することができます。
完全な Mizarアーティクルの文法（grammar）は Appendix B にあります。
以下のテーブルはMizarの予約語の全リストです。

according	aggregate	and	antonym
as	associativity	assume	asymmetry
attr	be	begin	being
by	canceled	case	cases
cluster	coherence	commutativity	compatibility
connectedness	consider	consistency	constructors
contradiction	correctness	def	deffunc
define	definition	definitions	defpred
end	environ	equals	ex
exactly	existence	for	from
func	given	hence	hereby
holds	idempotence	identify	if
iff	implies	involutiveness	irreflexivity
is	it	let	means
mode	non	not	notation
notations	now	of	or
otherwise	over	per	pred
prefix	projectivity	proof	@proof
provided	qua	reconsider	redefine
reflexivity	registration	registrations	requirements
reserve	such	scheme	schemes
section	selector	set	st
struct	such	suppose	symmetry
synonym	take	that	the
then	theorem	theorems	thesis
thus	to	transitivity	uniqueness
vocabularies	when	where	with
wrt			

いくつかの語は文脈に依存して異なった意味を持つことがあるので注意してください、例えば **for**（section 2.1 と 2.3.6 の使用法を比較してください）、**set**（section 2.2.3 の使用法と 2.3.3

の使用法を比較）あるいは **the**（section 2.3.3 での使用法と 2.3.5 での２つのはっきり異なった使用法を比較）などです。 さらに同義語（synonyms）の **be** と **being**、一方 **hereby** は **thus now** の省略表現（shorthand）として扱われます。 語 **according, aggregate, associativity, exactly, prefix, section, selector, to, transitivity** そして **wrt** は予約語ですが、現在までのところは処理系ソフトウェアに実装されていません。この言語は以下のような特別のシンボルを利用します：

```
,   ;   :   (   )   [   ]   {   }   =   &   ->
.=   ...   $1   $2   $3   $4   $5   $6   $7   $8   $9   $10   (#   #)
```

またMizarテキストではダブル・コロン（**::**）に出会ったときは１行コメントの始まりであることを心にとめておいてください。 概念の導入や数字の特別処理に使われるユーザ定義の識別子（identifiers）についてのそれ以上の情報はMizarのレキシカル・アナライザ（字句解析部）について述べた Section 3.1 で読むことができます。

2.1 式（Formulas）

Mizarは本質的に 1 階述語論理システムですから Mizar証明システムで文書にしてその正当性がチェックできるステートメントは古典的な論理連結子、および量化子と結合した原子（述語）式から構成されます。 しかし Mizarはすべての項（term）と結合した型情報（type information）を保持（maintains）するので、原子式（atomic formulas）は修飾式と属性式（qualified and attributive formulas）を持ちます（Mizarでどのような種類の概念が定義可能かは Section 2.3 を見てください）。

下のテーブルは標準的な論理結合子と量化子のMizar表記（representation）：

$\neg\alpha$	**not** α
$\alpha\wedge\beta$	α **&** β
$\alpha\vee\beta$	α **or** β
$\alpha\Rightarrow\beta$	α **implies** β
$\alpha\Leftrightarrow\beta$	α **iff** β
$\exists_x\alpha$	**ex x st** α
$\forall_x\alpha$	**for x holds** α
$\forall_{x;\alpha}\beta$	**for x st** α **holds** β

を示します。 実際にはそれぞれの量化変数（quantified variable）はその型を与えられなければなりません、それで量化子は：

```
for x being set holds ・・・
```

あるいは

```
ex y being real number st ・・・
```

という形をとります、ここで **set** と **real** は型の例を表します。　もしいくつかのステートメントが同一の型の変数を用いて書かれるのなら、Mizarは*予約（reservation）*でその選択された変数にグローバルの型を割り当てることを許します、例えば：

```
reserve x,y for real number;
```

と記述します。　これで量化式の中で **x** あるいは **y** の型を記述する必要はありません。予約宣言により、必要ならばMizarは暗示的（implicitly）に全称量化子を式に適用します。
　さらにユーザの便宜のために

```
for x holds for y holds ・・・
```

あるいは

```
for x holds ex y st ・・・
```

は、それぞれ

```
for x for y holds ・・・
```

と

```
for x ex y st ・・・
```

と短縮して書くことができます。　さらに、例えば

```
for x holds for y holds ・・・
```

あるいは

```
ex x st ex y st ・・・
```

と書くかわりにもっと便利な変数のリストを持つ形が許されます、すなわち

```
for x,y holds ・・・
```

と

```
ex x,y st ・・・
```

です。　ここで量化子の結合力は連結子のそれよりも弱いことにも触れておきます。

2.2　証明（Proofs）

Mizarアーティクルは、標準的な数学の論文と全く同様、導入された定義とそれについての記述（statement）からなり、これは証明に関してはテキストの最大の部分です。　そのデザ

インによりMizarは叙述型（declarative way）の書法をサポートし（すなわち、たいていは前方推論）、これは標準的な数学の慣習（practice）と似ています。 Mizar言語で書かれた証明は Jaśkowski の自然演繹法のスタイルの規則 **[6]**、あるいは F.B. Fitch **[3]** とK. Ono **[15]** により独立に開発されたそれに似たシステムに従って構成されています。 他のシステムに最も大きな影響を及ぼし、多数の手続きシステムの頂点に存在する同種の証明ソフトウェアの層を開発するという創造に対する刺激（inspiration）となったのは Mizar言語のこの部分です。 もっとも重要なものの名前を列挙すると、D. Syme による Declare システム **[18]**、J. Harrison による HOL のための Mizar mode **[5]**、M. Wenzel による Isabelle のための Isar 言語 **[22]**、F. Wiedijk による HOL-light のための Mizar-light **[24]**、そして最近の P. Corbineau による Coq のための declarative proof language (DPL) **[2]**などがあります。 証明を書くときの Mizarのやり方は F. Wiedijk により開発された ‘formal proof sketches’ **[23]** の概念モデルでもありました。 下に証明中のステートメントの構造を反映する典型的な*証明の骨格（proof skeletons）*を示します。 しかし、大部分の証明は複数の方法で述べることができます、なぜなら全てのMizarの式は内部的には単純化された “canonical form（正準形）” で表現されているからです（Section 3.2 *Mizar semantic correlates* の記述を見てください）。 証明の構造が証明された式の semantic correlates と合致する限り、それは正当なのです。 与えられた証明のスケルトンの正当性は Mizar verifier の REASONER と呼ばれるモジュールでチェックされます（Section 3.4を見てください）。

2.2.1 *証明の骨格（Proof skeletons）*。 どんな式 **Φ** についても、証明はその同じ式が最終結論として **thus** キーワードの後に記述されている証明ブロックの形をとります。実際は、これは原子式に対してのみ意味があります。

```
Φ
proof
 ...
 thus Φ;
end;
```

もし証明中の式が論理積命題（conjunction）ならば、証明は必ず２つの結論を含みます：

```
Φ1 & Φ2
proof
 ...
 thus Φ1;
 ...
 thus Φ2;
end;
```

含意命題を証明するなら最初に前項を仮定し後項で結論するのが最も自然な証明です

```
Φ1 implies Φ2
proof
 assume Φ1;
 ...
 thus Φ2;
end;
```

Mizarは同値命題を２つの含意命題の論理積命題と解釈するので、次のような証明のスケルトンになります。

```
Φ1 iff Φ2
proof
  ...
  thus Φ1 implies Φ2;
  ...
  thus Φ2 implies Φ1;
end;
```

この場合、次のスケルトンを使えば、証明のネスティングの深さを少し減らせます。

```
Φ1 iff Φ2
proof
  hereby
   assume Φ1;
   ...
   thus Φ2;
  end;
  assume Φ2;
   ...
  thus Φ1;
end;
```

論理和命題を証明する典型的な方法は最初の論理和命題が成立しないと仮定して、他の命題を証明することです。

```
Φ1 or Φ2
proof
 assume not Φ1;
```

```
 ...
 thus Φ2;
end;
```

論理的な観点からは、同様に２番目の論理和命題が成立しないと仮定し、最初の論理和命題の正当性を示すことで証明を完成することもできるはずなのですが、ユーザは証明のスケルトンが無効であるという警告をうけます。 それは REASONER が使用するsemantic correlates のレベルでは論理積命題（conjunction）は可換（commutative）ではないからです（3.4 を見てください）。

さらに、どの式も以下の間接証明法のように *reductio ad absurdum* method *（背理法）*を使って証明可能であることにも留意してください。

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

全称命題式の証明は恣意的に選んだ、しかし固定のある型の変数を選択することからスタートし、次にそれを代入した式の正当性を結論します。

```
for a being Θ holds Φ
proof
 let a be Θ;
 ...
 thus Φ;
end;
```

もし a が予約された型 Θ なら'be Θ'と'being Θ'のフレーズは不必要なので除去できることに留意してください。 存在ステートメントの証明では、証人となる項(witness term) a とそれにみあう適切な結論を示さなければなりません。

```
ex a being Θ st Φ
proof
  ...
take a;
 ...
 thus Φ;
```

```
end;
```

a の型は量化子のなかで使われる型とフィット（Mizar のジャーゴンでは：*widen to*、*拡張される*）しないといけないのは明白です。

2.2.2 *ジャスティフィケーション (justification)*。 Mizarは疑う余地のない概念である推論の自明性（Section 3.5 を見てください）が装備されている CHECKER モジュールを使ってアーティクルのなかのすべての 1 階述語論理ステートメントの論理的正当性をチェックします。 もしステートメントが CHECKER にとって自明である推論を生成するなら、セミコロン（;）での終了で十分で CHECKER はエラーを出しません。 しかしながら、ほとんどの場合、なんらかの適切なジャスティフィケーション（正当化、justification）の不足があり、それは Mizarテキストの中に次のようなエラー・フラグが挿入されることで告知されます。

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

この ***4** マーカーは一つ前の行の中で欠落しているジャスティフィケーションの位置を示しています。 CHECKER はユーザが推論をジャスティファイするまでエラーを出し続けます。 上に示したようにジャスティフィケーションはキーワード **proof** で始まり **end** および終了（closing）のセミコロン（**;**）で終わります。 その行以前に記述されたステートメントの帰結として CHECKER により明らかである（obvious）と判明した証明は **by** キーワードとそれにつづくコンマで区切られた複数の適切なステートメントへの参照のリストからなる *straightforward justification* に還元され、セミコロンで終ります。 参照はそれぞれのラベルがつけられたステートメントを指します。 例えば以下に示すように。

```
lb1: Φ1;
...
lb2: Φ2;
...
 Φ by lb1, lb2;
```

証明はネスト（入れ子構造に）することもできるので、いろいろな証明レベルの式を以前に定義された証明のブロックをオーバーライドしてマークするのにラベルを使うこともできます。

（ローカルあるいは MML から取り出せる）分離した Mizarアーティクル中の定理への参照を使うことも可能です。 その時、参照は **<FILENAME>:<number>** あるいは **<FILENAME>:def <number>** の形式をとります、ここで **<FILENAME>** は参照されるアーティクルで **<number>** は定理あるいはそのアーティクルのなかの定義に対応します。

有効なラベルの定義とファイル識別子については Section 3.1 を見てください。

証明のステップについては線形推論の経路（linear reasoning path）がきわめて典型的なものであることに注意してください。 このことをおしすすめるために（to facilitate this）

```
lb1: Φ1;
...
lb2: Φ2;
Φ by lb1, lb2;
```

ではなく

```
lb1: Φ1;
...
Φ2;
then Φ by lb1;
```

という具合に **then** とリンクしたステートメントを書くこともできます。

類似の方法で、結論へ至る線形の推論は **thus** と **then** の代わりに **hence** キーワードで閉じることもあります。

ステートメントがスキーム（２階述語論理の自由変数）を使ってジャスティファイされる場合、単純ジャスティフィケーションは **by** ではなく **from** キーワードを利用します、それ（fromキーワード）は Mizar verifier に推論の正当性のチェックに別のモジュール SCHEMATIZER を使うべきだと指示します。 その場合、参照はある番号がアーティクル **<FILENAME>** に含まれている場合には **<FILENAME>:sch <number>** という形をとり、ローカルのスキームに対しての参照の場合は識別子になります。 スキームは通常スキームの前提を持つので、完全なスキームの参照は終了のセミコロンの前に、カッコの中に置くべきスキームの前提に代入するための、コンマで分離された式への参照のリストが必要です。 **by** の後の straightforward justification の参照と異なり、ラベルの記述の順番は無関係なことに注意してください。 それ以上のMizarのスキームといくつかのサンプルの情報については Section 3.5.1を見て下さい。

2.2.3 *補助的な証明の要素（Auxiliary proof element）。* Mizarの証明の基本構造を導入しステートメントをジャスティファイする方法を手に入れても、すべての Mizarの証明を完全に理解するには証明を構成する他の Mizarキーワードの意味論を勉強しないといけません。 以下に手短にこれらの補助的な証明の要素とその使用法の例を示します。

本当はなくても済む２つの構文から始めましょう、でもそれらの使用は数学の慣用表現に良く似ています。 ひとつは下に示す形の *iterative equality（反復等号）*です。

α_1 = α_2 *justification*

.= α_3 *justification*
...
.= α_n *justification;*

ここでα_iは項です、そして.=を使わない以下のような形に変換されます。

α_1 = α_2 *justification;* **then**
α_1 = α_3 *justification;* **then**
...
α_1 = α_n *justification;*

上の一続きの式は反復等号の全体と等価です。

他の構文はフレンケル演算子（Fraenkel operator）、すなわち ZF（Zermelo-Fraenkel 公理系）における集合の内包表記（set comprehension）バージョンです。（訳注：反意語はset enumeration 列挙表記？、あるいは connotation 内包 ⇔ denotation 外延 か？）フレンケル項の正確な識別のために、**SUBSET**を **requirements** 環境指令に追加します。

フレンケル演算子の一般的な形は以下のようです：

$$\{t \text{ where } v_1 \text{ is } \Theta_1, \ldots, v_n \text{ is } \Theta_n : \Phi\}$$

ここで ***t*** は項、v_1 は Θ_i 型の変数です。 全ての **i** について型 Θ_i は **Element of A** まで拡張されねばなりません、そこでは **A** は恣意的に選べます、必ずしも非空集合（non-empty set）である必要はありません、そして **Φ** は式です。 さもないとエラー ***129** が報告されます。 変数のリストは ***t*** と **Φ** にあらわれる必要はありません。

この式の最も普通の展開は以下のように与えられます。

```
  x in { t where t is Element of A : not contradiction }; then
  consider t1 being Element of A such that
A1: t1 = x;
```

set というキーワードはより複雑な項の省略型の導入に使うことができ、タイピングを簡単にします。

```
  set A = X \/(Y /\ Z);
  set B = (X \/ Y) /\ (Y \/ Z);
  A = B by XBOOLE_1:24;
```

そして **A** と **B** は以後の証明の残りの部分で使うことができます。

reconsider（キーワード）は与えられた項の型が、あたかも定まった型（stated）であるかのように扱うことをシステムに強制します； 通常これは特定の型がある構図（例えば定義の拡張）で要求された場合などに使われ、項が証明の中に導入された後で項がこの型を

もつという事実についての追加の推論（extra reasoning）が要求されます。

```
  let k, n be Net;
  assume k >= n; then
  reconsider m = k - n as Element of NAT by NAT_1:21;
```

consider（キーワード）は規定された（stated）条件を満たす与えられた型の新しい定数を導入します。

```
  consider k being Integer such that
A1: m = 2 * k + 1;
```

それは存在量化子の支配下の変数にアクセスする方法でもあります。

```
  ex k being Integer st x = 2 * n; then
  consider n being Integer such that
A1: m = 2 * k1 + 1 & k1 > 0;
```

この構図（construct）に関しては等位項（conjuncts、訳注：&で結合された項）の順序が重要であるのを観察してください； もし、**A1** ラベルにある論理積命題（conjunction）の（訳注：各項）を入れ替えようとすると ***4** エラーをもらいます。

given（キーワード）は、後ろに **consider** が続く存在のステートメント（**ex**）の仮定（**assume**）の短縮です、なので

```
  assume ex n being Integer st x = 2 * n; then
  consider n being Integer such that
A1: x = 2 * n;
```

の代わりに

```
  given n being Integer such that
A1: x = 2 * n;
```

と書くほうが短いのです。

thesis（キーワード）は*証明されねばならないフレーズ (*phrase *which has to be proved)* の役割を演ずるキーワードです； なのでダイナミックに変化する命題（thesis）を置き換えて証明の最後のステートメントを有意に短縮します; これは proof の中でだけ許されます、それ以外では ***65** エラーが報告されます。

contradiction（キーワード）は論理的には*偽*（*falsum*：ラテン語）と読むことができます。 それはしばしば否定の文脈で使われます； とても頻回に **not contradiction** がフレンケル演算子のなかに現れるように。 これは矛盾による証明のなかでは thesis の役割を

演じます、その場合 **thesis** と **contradiction** は交換可能だからです。

per cases（キーワード）は取り尽くしによる証明（proving by exhaustion）を進めます。もしも cases のリストが論理恒真式という点から見て（通常の場合は排中律からみて）全てを尽くしているならば、それは自身の証明を必要としません、そうでないときそれは参照のリストによってジャスティファイされねばなりません（ここではネストした proof は許されません）。 この構文は終止の end を必要としません。

suppose（キーワード）は取り尽くし法による証明（**per cases**）で新しい case を始める言葉です。 命題は変化しません、すなわち全ての case で同一です。 以下に２つの case を持つ単純な証明を提示します：

```
reserve A,B,C for set;
theorem
  A / B c= implies A c= B \/ C
  proof
     assume
A1: A \ B c= C;
     let x be set;
     assume
A2: x in A;
     per cases;
     suppose x in B;
       hence thesis by XBOOLE_0:def 3;
     end;
     suppose not x in B;
       then x in A \ B by A2, XBOOLE_0:def 5;
       then x in C by A1,TARSKI:def 3;
       hence thesis by XBOOLE_0 def 3;
    end;
end;
```

theorem キーワードがステートメントを輸出可能（exportable）にしているのに注意してください（これにより他のアーティクルからそれを参照することができるようになります）。

case（キーワード）は **suppose**（キーワード）に類似しています； たとえ後者がcase による証明のアイデアを忠実に（faithfully）反映していたとしても case を使うことができます、例えば等位項（conjuncts）の論理和（disjunction）の証明に case を使うことができます。 **suppose** と **case** の両方とも終止の **end** と（訳注：対にして）囲むようにしま

す。

now は語数の多い冗長な推論（diffuse reasoning）を開始します（これは終止の **end**を必要とします）。　証明されるステートメント（proved statement）は明示的には書かれていません、REASONER にステートメントをダイナミックに再構成させるようになります ー これは一般化変数と長い命題の長いリストの場合に有用です。

一般的な形は

```
now
  ...
  thus Alpha(x);
end;
```

で、ここでドット（ . . . ）は推論（式の一続き）（reasoning (sequence of formulas)）に置き換えられます、そして単なる **Alpha(x)** の証明になります。　通常は恣意的な diffuse statement の式への変換には問題がありません、たぶん

```
now
  assume Alpha;
  ...
  take x;
  thus Beta;
end;
```

を例外として（maybe with the exception of ）。　ここで例証（exemplification）は次の式に変換しもどすことができます。

```
  Alpha implies ex x st Beta;
```

hereby（キーワード）は **thus** と **now** の短縮の一種として取り扱うことができます。この構文で証明のネスティングを一つ除去することが許されます。

```
hereby
  assume k in n;
  ::here the proof...
  hence k < n;
end;

k in n implies k < n
proof
  assume k in n;
```

```
::here the proof...
  hence k < n;
end;
```

2.3 概念の定義（Defining notions）

このセクションで Mizar言語での新しい概念の定義の仕方をお見せします。　どういった種類の概念を導入することができ、どのようにそれらが標準的な‘ペンと紙の’数学で使われているものに対応するのかを知ることは重要です。

始めに、次の式を考えてみましょう：

$\forall_{n \in \mathbb{N}}$ n + 1 > 0

どの自然数でも、1との和は正である

for n being a natural number it holds that n + 1 > 0

n を自然数とするとき n + 1 は正である

上の式は全て同じ性質を表しています、しかし異なった統語－論理手段（syntactic and logical devices）を使っています。　最初の式には全称量化子があり、∀ あるいは **"for** … **holds** …**"** と記述されています。 ひとつの変数 n と数字の 0 と 1 があり、それは **+** 演算シンボルを使って項を構成します。　n が使われるときはいつでも、それがどの種類の値を取ることができるのか、べつの言葉ではその n の型は何なのか（この場合は ‘natural number’、「自然数」です）を常に記述します。

型は単一の名詞（‘数’、‘number’）から、あるいは先頭につけるある形容詞（‘自然’、‘natural’ のような）を追加してより正確にして、ステートメントが真になるように作成されます。　上の式には述語式（predicate formula）を作成するのに使われる関係のシンボル > があります、そして ‘positive’ という形容詞を使って作成された属性式（attributive formula）もあります。

これらのすべての構成（construct）は Mizarでサポートされます。　すなわち、*述語（predicates）*は式を作るために使われ、*mode* は型（type）のコンストラクタ（構築子、constructor）で、形容詞（adjectives）は*属性（attributes）*を用いて構築され、そして*ファンクタ（functors）*は項のコンストラクタです。

全ての新しい概念は **definition** キーワードで始まる*定義ブロック（definitional block）*内で導入され、対応する **end;** で終了します。　そのブロックの中では：

— （もし必要ならば）引数（複数）とそれに対応した型、それらは場所（訳注：局所変数）（loci、単数はlocus）と呼ばれ **let** キーワードにより導入されます。

— おそらく **assume** あるいは **given** の後に続く定義のためのある仮定

— 定義の主要部分 – この部分の詳細な記述は次の section に続きます

— もし定義の健全性（**soundness**）を保証することが要求されるのなら正当性条件（**correctness conditions**）

— 該当する場合には（if applicable）導入された概念の属性（**properties**）（例えば演算の可換性（**commutativity**）、あるいは関係の反射関係（**reflexivity**））

— さらに、定義ブロックは現在のブロックのなかのいくつかの証明で使われるかもしれない局所的推論を含むかも知れません。

単一の定義ブロックが多数の概念を導入するために使われることもあります。 しかし新しい概念は **end;** でブロックが閉じたあとで*使用可能*になります。 例えば以下のテキストでは Mizarシステムは未知の述語を意味する ***102** のフラグでマークしたエラーを報告します。

```
definition
  pred_symbol_1 means not contradiction;
  pred_symbol_2 means pred_symbol_1;
::>                                *102
::> 102: Unknown predicate
end;
```

この問題を解決するには、分離した定義ブロックで後者の述語を導入しなければなりません：

```
definition
  pred pred_symbol_2 means pred_symbol_1;
end;
```

さて局所変数の宣言に集中しましょう。 まず初めに定義ブロックのなかの宣言部においては、どの引数も省略されません。 例えば、部分集合の全体集合についての補集合を定義するとき、両方の引数とも **let** キーワードで開始された局所変数セクション内で宣言されねばなりません。

```
definition
  let E be set, A be Subset of E;
  func A` -> Subset of E equals
  E \ A;
  ...
end;
```

そのため以下の定義は間違いです。

```
definition
  let A be Subset of E;
::>                   *140
  func A` -> Subset of E equals
  ...
end;
::> 140
::> Unknown variable
```

すでに変数 **E** をアーティクルのなかで局所的に定義していてもエラーが報告されます。

```
set E = the set;
...
definition
  let A be Subset of E;
::>             *222
  func A` -> Subset of E equals
  ...
end;
::> Local constants are not allowed in library items
```

補集合をとることは実際には２引数の演算なのですが、上のような定義は１つの*隠れた引数*（**E**）と*１つ可視の引数*（**A**）となります。　もうひとつ別の定義は以下のようになるでしょう：

```
definition
  let E,A;
  func `(E,A) -> Subset of E equals
  E \ A;
  ...
end;
```

ここでは両方の引数とも可視です。　でも標準的数学の演習でもそうですが、この定義が頻回に使われることはありません。　しかしMizarシステムは最も少ない引数を使うべきだという要求はしません。　ですが（yet）引数を繰り返すことはできません。

```
definition
  let E,A;
  func `(E,A,A) -> Subset of E equals
```

```
  ::>        *141
  E \ A;
  ...
end;
::> 141 Locus repeated
```

Mizar局所変数（Mizar loci）のカギとなる特性はどの局所変数も必ずもう一つの局所変数のパラメターか、あるいは定義された概念の可視引数として使われていることです。　われわれの例の定義では、集合 **E** はその部分集合 **A** の型のパラメターです、ですからそれは可視の引数である必要はありません。　しかし、次の定義は間違いでしょう（would be incorrect）：

```
definition
  let E be set, A be Subset of E;
  func 'E -> Subset of E equals
  ::>  *100
  E \ A;
  ...
end;
::> 100 Unused locus
```

なぜなら **A** はその要件に従わないからです。

この論文の全体を通じて、読者の便宜のために定義された概念の記述の文脈では必ずしもその完全な意味を与えてありません、*フォーマット*と呼ばれる「定義された概念のシンボルとともに左引数と右引数の数」を与えれば十分でしょう。　フォーマットに全ての局所変数の型情報（もし該当するなら結果の型も）を付け加えれば、定義の*パターン*を得ることができます。

その術語の一般的セットを用いて、ある種のコンストラクタ（構築子、constructors）の定義に特有な面に進みましょう。

2.3.1 *述語（Predicate）*。　述語の定義は、他の全てのコンストラクタと同様に、定義ブロック内で記述されます。　局所変数リストの宣言の後の **pred** キーワードの後に定義の述語の*フォーマット*が続きます。　引数の個数（arity）は述語シンボルの両方の側の局所シンボルの数で指定されます。

```
definition
   let f,g be Function; let A  be set;
   pred f,g equal_outside A means
:Def:
```

```
  f|(dom f \ A) = g|(dom g \ A);
end;
```

ここの **equal_outside** という述語は３つの引数を持ちます：２つの左引数と１つの右引数です。　その後、あとでその定義を参照するために使うラベル（オプションです）の後に **means** キーワードが続き、デフィニエンス（定義するもの：*definiens*）すなわち定義された関係を記述した式が続きます、われわれの例では： **f|(dom f \ A) = g|(dom g \ A)** です。

定義への参照は暗黙のうちに、システムによって作成された*定義定理（definitional theorem）*への参照を意味します、われわれの例ではそれは以下のような意味を持ちます：

```
for f,g being Function, A being set holds
f,g equal_outside A iff f|(dom f \ A) = g|(dom g \ A);
```

Mizar言語は“段階的定義（piecewise definitions）”もサポートします、そこではセッティングが異なると実際の意味も異なります、例えば：

```
definition
  let x,y be ext-real number;
  pred x <= y means
  ex p,q being Element of REAL st p = x & q = y & p <= q
    if x in REAL & y in REAL
  otherwise x = -infty or y = +infty;
end;
```

その場合、定義定理は以下のような形になります。

```
for x,y being ext-real number holds
(x in REAL & y in REAL implies
  (x <= y iff ex p,q being Element of REAL st
          p = x & q = y & p <= q)) &
(not x in REAL or not y in REAL implies
  (x <= y iff x = -infty or y = +infty));
```

定義された概念のあるものはその定義ブロックのなかに特有の*特性*（specific *properties*）の証明とステートメントを伴うことができます。　例えば、常に対称関係が成立するように定義された関係はその対称性（**symmetry** property）を定義のなかに持つ形式で記述されるという事実があります。　そのあとでは Mizarベリファイヤ（Mizar verifier）は、その概念が使われるときはいつでも、付加的な発見的問題解決法としてこの特性を使うことができます。　Mizarがサポートしている特性についてのもっと多くの情報を Section 2.5 で

見ることができます。

さらに、いかなる Mizarの定義も **assume** キーワードに続くある種の仮定の下で形式化されていることを述べましょう。 典型的には、仮定は導入された概念がうまく（矛盾をはらまずに）定義されている（well-defined）ことを保証するのに必要とされます、すなわち要請された正当性条件 (correctness condition) は与えられたデフィニエンス (definiens：定義するもの）について証明されます。 この言語で（訳注：使用が）否認されていないにもかかわらず、述語の定義のなかで仮定を使うのはほとんど道理にかないません。 なぜならそのような場合正当性の条件は要求されないからです、ですからその仮定は不必要に制限された定義という結果になってしまいます。

2.3.2 *属性（Attributes*）。 *属性*は形容詞のコンストラクタです。 属性は **attr** キーワードで定義することができ、後にその*主語（subject）*、**is** キーワード、属性のシンボル、そして **means** キーワードの後にデフィニエンス（マーキングで参照できるようにするためのラベルをデフィニエンスの前につけることもできます）が続きます。 主語の役割を演ずる局所変数は全ての局所変数のリストの最後のものであることが要求されます。 さもないと、定義のなかで使用されない局所変数が存在してしまうかもしれません。

```
definition
  let E be set, A be Subset of E;
  attr A is proper means :Def:
  A <> E;
end;
```

CHECKER が使う対応する定義定理は、定義に対する参照がおきたときはいつでも下のように形式化されます：

```
for E being set, A being Subset of E holds
A is proper iff A <> E;
```

Mizarの属性システムの重要な拡張は*可視引数を持つ属性（attributes with visible arguments）*のサポートです。 このメカニズムは例えば、**n**-次元空間の概念あるいは **r**、**c-sized** マトリックス（実数や複素数サイズの行列）の概念の導入を可能にします。 その場合、記法（syntax）は **is** キーワードの後で、しかし属性シンボルの前で複数引数のリストを認めます、たとえば：

```
definition
  let N be Cardinal, X be set;
  attr X is N-element means
  card X = N;
```

```
end;
```

可視引数を持たない属性の場合と全く同じように、システムは主語が最後の局所変数であると指定されることを要求します。　上の例では全ての局所変数（**X** と **N**）がパターンのなかで使われているにもかかわらず、主語（**X**）の位置が決定されます。　それゆえ次の定義は間違いです。

```
definition
  let X be set, N be Cardinal;
  attr X is N-element means
::>     *374
  card X = N;
end;
::> 374: Incorrect order of arguments in an attribute definition
```

2.3.3 *モード（Modes）*。　Mizarの型は*モード*を使って構成します。　Mizarは２種類のモード定義をサポートします：

— 前もって定義されそれに適用できる基底型（*radix type*）を伴う形容詞の集合体（collection）（*クラスタ*と呼ばれる）として定義されたモードで、拡張可能モード（*expandable mode*）ともよばれる

— 明示的なデフィニエンス（定義するもの）を持つ型を定義するモード、デフィニエンスはオブジェクトがその型をもつように記述（fulfilled）されなければならない。

拡張可能モードは **mode** キーワード、オプションの引数リストを持つモードのシンボル（引数は **of** キーワードの後に書きます）、**is** キーワードが続き、形容詞と基底型のリストがつづきます。　例えば：

```
definition
  mode Function is Function-like Relation;
end;
```

名前が示すように拡張可能モードはただのショートカットあるいはマクロで、システムにより内部的に基底型を伴う形容詞の集合体に展開されます。　**Function** 型のオブジェクトは **Function-like Relation** 型も持ちます、それゆえ検証システムはそれを受け入れる（accept）のにいかなる参照も必要としません、これはその逆よりも良いのですが（the other way round）。

非拡張可能モードの定義は記法（syntax）が少し違います、なぜなら基盤をなす特性（property）は定義に明示的に記述してあるからです。　そこで **mode** キーワード、シン

ボル、もしあれば（potential）引数のリストの後で、矢印 **->** と、その後に **means** キーワードとデフィニエンス（ラベルが付いているかもしれません）がつく*母型（mother type）*があります。 例えば：

```
definition
  let X be set;
  mode a_partition of X -> Subset-Family of X means :Def:
  union it = X &
  for A being Subset of X st A in it holds A <> {} &
    for B being Subset of X st B in it holds A = B or A misses B;
end;
```

デフィニエンスのなかの **it** キーワードが定義された型を保持するオブジェクトを参照するために使われているのに注意してください。

Mizarの型システムの鍵となる特徴の一つは導入された型が non-empty でなければならないことで、与えられた型のオブジェクトが少なくとも一つは存在しないといけません。この制限は形式化された理論は常にある外延／明示的意味（denotation）を持つことを保証するため導入されました。 それゆえモード定義は存在条件（**existence** condition）の形で記述された事実（fact）の証明を要請します。 正当性条件（correctness condition）については、これ以上の情報は Appendix A にあります。 拡張可能モードにおいては、これと等価な存在条件は形容詞のクラスタの存在登録（existential registrations）により実現されています。 そこで、最初に登録しておかないといけませんが、その表現の前に **Function-like Relation** を新しい拡張可能モードを定義するために使うこともできます（Section 2.6 を見てください）。

型が空でないこと（non-emptiness）は推論（reasoning）に **consider** ステートメントを使ってあらゆる型の変数の導入を可能にすることに注意しましょう、なぜならそれらの存在は保証されていますから。 さらにMizarは内部的に、構造を特定することなく全ての型につき１つのオブジェクトを生成するという意味で、*全域での選択（global choice）*をサポートすることができます、例えば：**the real number** は固定の実数です。 術語 **the real number** が使われる時は、その値が特定されていなくとも、いつでもそれは同一の数なのです。

Mizarの型の木（tree）（型の拡張関係のことです）のなかで定義されたモードの直接の祖先を指定（specify）するため、母型がモードの定義の中で使われます。 新しいモードで構築（construct）された型はその母型まで*拡張（widens to）*されます。 この拡張関係は型に伴う形容詞も計算に入れます、すなわちより短い（あるいは同程度の）形容詞のリストを持つ型はより広くなるとみなされます。 初めから組み込んである型 **set** はいつの場合も最も広い型です。

上に述べた定義の例で、システムは自動的にどの区分（partition）も基礎となる集合（underlying set）の部分集合の族であることを正しいとして受け入れます。 それは部分集合の族に対して定義された全ての概念が、区分についても適用できることを意味します。しかし、ある部分集合の族が実は一つの区分であると記述するためには、対応する定義定理への参照が必要です。 上の定義のためにシステムにより生成された定理は以下のような意味を持ちます：

```
for X being set, IT being Subset-Family of X holds
IT is a_partition of X iff
union IT = X &
for A being Subset of X st A in IT holds A <> {} &
   for B being Subset of X st B in IT holds A = B or A misses B;
```

非拡張型モードは再定義ができます（Section 2.4を見てください）が、拡張可能型モードはそのマクロに似た性質（macro-like nature）のため再定義ができないことに注意してください。

2.3.4 *ファンクタ（Functors）。* Mizarにおける項のコンストラクタ（constructors）はファンクタと呼ばれます。 すでに定義した他のコンストラクタの場合とまったく同様に、ファンクタの定義は（複数の）局所変数の宣言と（もし必要なら）仮定ではじまります。

それから **func** キーワードの後で、ファンクタのフォーマットが指定されます。 Mizarはプレフィックス、インフィックス、そしてサフィックス・フォーマットをサポートし、同様に区間（intervals）、その他の概念をコード化する時有用な両面接辞（circumfix）（カッコのようなもの、訳注：[ ]や( )など）もサポートします。

もしファンクタのシンボルの左あるいは右側の引数の数が 1 以上なら、これらの引数はカッコの中に書かないといけません。 フォーマットの後に矢印 **->** があり、定義された演算の型がそれに続きます。 型が **set** の場合（全てのオブジェクトが自動的に持つ最も一般的な型です）、このフレーズは省略することができます。 最後に項の意味が **means** キーワードの後に与えられ（通常ラベルの付いた）デフィニエンスが続きます、例えば：

```
definition
  let X,Y be set;
  func X \/ Y -> set means
  x in it iff x in X or x in Y;
  existence;
  uniqueness;
end;
```

キーワード **it** が、宣言された型とともに、定義された項を参照するためデフィニエンス内で使われるでしょう。

ファンクタはもしその定義が **existence** と **uniqueness** の２つの正当性条件の証明を持つなら well-defined になります。 正当性条件として証明される式についてのこれ以上の情報については Appendix A を見てください。

他の定義の場合のように、定義への参照は実際にはその定義定理への参照になります。私たちの例では次のようになります：

```
for X,Y,Z being set holds Z = X \/ Y iff
for x being set holds x in Z iff x in X or x in Y;
```

ファンクタのデフィニエンス（定義する項）がしばしば **it = ...** という形をとる場合があります。 言いかえると、定義されたオブジェクトは構成されている（constructed）ある項と単純に等しいということです。 そこで、そのようなファンクタを定義するには、ちょっとだけ違った記法(syntax)を使うというのがお勧めです(advisable)、つまり **means** キーワードと標準的なデフィニエンスを使わないで、**equals**（キーワード）を使いその後にそれと等しい項である新しいオブジェクトを続けて記述します、例えば、対称差（symmetric difference）は以下のように定義されます：

```
definition
  let X, Y be set;
  func X \+\ Y -> set equals
  (X \ Y) \/ (Y \ X);
  coherence;
end;
```

その場合、要求される正当条件は違うものになります（**coherence** になります）。 すなわちその項がすでに構築されているならば、それは明らかに存在し、かつユニークです、しかし、その項が宣言された型をもつことを証明する必要がありそうです。

その場合、定義定理は簡単な形になります：

```
for X,Y,Z being set holds
Z = X \+\ Y iff Z = (X \ Y) \/ (Y \ X);
```

定義定理は常にシステムにより生成されその概念に関する証明のなかで参照されるのにもかかわらず、しばしば参照する必要がないことがあります。 なぜなら CHECKER が*同値拡張（equals expansion）*を使い、そういう項（われわれの例では **X \+\ Y** のようなもの）があらわれるたびに自動的にその簡約前の完全な形（この場合 **(X \ Y) \/ (Y \ X)** ）に展開するからです。 この展開は **definitions** 環境指令のなかに、その定義を含むアーテ

ィクルの名前がリストアップされると発生します（Section 4.4 を見てください）。 現行Mizarシステムの実装ではこの展開は 3 レベルまでに制限されていることに注意してください。

2.3.5 *構造体（Structures）*。 Mizarの構造体は群、位相空間、圏（categories）、その他の数学的概念をモデル化するときに使用され、それらは通常、組（tuples）として表現されます。 それゆえ、構造体の定義はそのフィールドを示す*セレクタ（selectors）*のリストを含みその名前と型で特徴づけられます、例えば：

```
definition
  struct multMagma
  (# carrier -> set,
     multF -> BinOp of the carrier #);
end;
```

ここで **multMagma** は 2 つのセレクタを持つ構造体の名前です： それの台集合（**carrier**）と呼ばれる恣意的に選ばれた集合とそれに働く **multF** と呼ばれる 2 項演算子です。 この構造体は群を定義するために使うことができますが、上束と下束（upper and lower semilattices）の定義にも使えます、実際には集合とそれに作用する 2 項演算に基礎をもつあらゆる概念に使用可能です。

上にあげた構造体はまだ群を（そしてもっと具体的なオブジェクトも）定義しているわけではないのに注意してください、なぜなら **multF** の特性についての情報が全くないからです。 構造体はただ理論を発展させるための基礎構造でしかありません。 実用上は、要請される構造体を導入した後で、さらに一連の属性があるフィールドを記述するために定義されます。

まえに述べたように、上の **multMagma** 構造体は群の概念の定義だけではなく他の概念の定義でも使われます。 それでもなお、その構造体での演算は **multF** の名前を継承します、なぜなら現行のMizarの実装ではセレクタに対する（あるいは構造体全体に対する）同義語導入のメカニズムを提供していないからです。 それゆえ、標準的な数学演習で異なった名前が頻回に使用される場合には別の構造体を導入するほうが良いでしょう。 例えば、束の演算は通常 **meet** と **join** と呼ばれます、そこで下束（lower semilattice）は：

```
definition
  struct /\-SemiLattStr
    (# carrier -> set,
       L_meet -> BinOp of the carrier #);
end;
```

というふうにより良いコードに変換されます。 Mizarは構造体の多重継承（multiple

inheritance）、それは Mizarライブラリの中で相互関係のある構造体の全体的な階層構造を利用可能にするのですが、をサポートするので **1-sorted** 構造体をほとんどすべての他の構造体の共通の祖先とすることができます。　例えば位相群（topological group）のMizarによる形式化は、お互いに独立に定義し発展させた群論と位相空間論の２つの理論を新しい構造体の基礎のうえに一緒にマージすることで可能です。　例えば：

```
definition
  struct (1-sorted) TopStruct
  (# carrier -> set,
     topology -> Subset-Family of the carrier #);
end;

definition
  struct (multMagma, TopStruct) TopGrStr
  (# carrier -> set,
     multF -> BinOp of the carrier,
     topology -> Subset-Family of the carrier #);
end;
```

このアプローチの利点は、群と位相空間に関するすべての概念と事例（notions and facts）が自然に位相群に適用可能になることです。

新しい構造体を導入するときに、継承したセレクタはそれらのあいだの関係が保存される限りどのような順番で並べてもよいことに注意しましょう。　先祖（継承元）の構造体の名前は定義される構造体の名前のまえのカッコの中におかれます。　Mizar構造体システムの重要な拡張の一つは、恣意的な集合あるいは他の構造体を媒介変数とする構造体を定義することが可能なことです。　例えば：

```
definition
  let F be 1-sorted;
  struct (addLoopStr) VectSpStr over F
  (# carrier -> set,
  addF-> BinOp of the carrier;
  ZeroF -> Element of the carrier,
  lmult -> Function of [:the carrier of F,the carrier:],
           the carrier #);
end;
```

例えば、上記の構造体は実数体あるいは複素数体上のベクトル空間の基礎として使うことができます。　システムは数個のパラメターを持つ構造体を定義することを許しますが、

それらは全て定義の中の局所変数として導入されなければなりません。

具体的な数学のオブジェクト、例えば整数の加群（additive group of integers）はいわゆる*凝集体（aggregates）* －構造体の定義により自動的に定義される特別な項のコンストラクタ、例えば： **multMagma(#INT,addint#)**－ と一体となって導入されます、ここで**INT** は整数の集合で **addint** は加算の機能（addition function）を表します。

凝集体で使われた全ての項は構造体の定義のなかでの宣言に相当するそれぞれの型を持つ必要があります。 私たちの例では **INT** は明らかに集合型（set 型）で、**addint** は **BinOp of INT** 型でなければなりません。

どの構造体も暗黙的に特別の属性 **strict** を定義しています。 相当する形容詞（訳注：strict）の意味は構造体型のオブジェクトはその構造体のために定義されたフィールド以外は、なにも含まないということです。 例えば、**TopGrStr** に基づく構造体型を持つ項は **strict TopGrStr** でしょう、それは **strict multMagma** でもないし **strict TopStruct** でもないということです。 構造体の凝集体を使って構成されたそれぞれの項は明らかに **strict** です。

最後に、Mizar言語は複合的な構造体型を持つ与えられた項をその well-defined サブタイプ（well-defined subtype）に制限する手段を持っています。 この特別の項のコンストラクタ、*忘れっぽいファンクタ（forgetful functor）*もまた構造体の名前を利用します、例えば **multMagma of G**、ここで **G** は **multMagma** 構造体から継承した潜在的により広い型をもちます。 繰り返しますが、そのような項は与えられた構造体型に関して **strict** です。

2.3.6 *同義語と反意語（Synonyms and antonyms）*。 もしオリジナルのシンボルの代わりに別のシンボルを使わなければならない場合にはMizarは関係（relation）、形容詞（adjective）、型（type）、あるいは演算（operation）についての同義語の使用をサポートします。 新しいシンボルは、もちろん、ボキャブラリ・ファイルに追加されねばなりません。 同義語は定義ブロックに類似の **notation ... end;** ブロックのなかで導入されます。 例えば：

```
definition
  let A be set, B be Subset of A;
  pred pred_symbol_1 A,B means not contradiction;
end;

notation
  let A be set, B be Subset of A;
  synonym pred_symbol_2 A,B for pred_symbol_1 A,B;
end;
```

同義語の宣言内の局所変数の型はオリジナルの定義の中のそれよりも狭くないといけません。　しかし、局所変数を定数項で置き換えることは不可能です、ですから集合 **A** を自然数の集合 **NAT** で置き換えることはできません。

```
notation
  let B be Subset of NAT;
  synonym pred_symbol_2 B for pred_symbol_1 B,NAT;
::>                                          *300,142
::> 142: Unknown locus
::> 300: Identifier expected
end;
```

同義語はフォーマットを変更するのにも使われます、例えばプリフィックスをインフィックスにする下の例のように：

```
notation
  let A be set, B be Subset of A;
  synonym A pred_symbol_2 B for pred_symbol_1 A,B;
end;
```

さらに同義語は可視引数の数を変化（増加あるいは減少）させます、例えば：

```
notation
  let A be set, B be Subset of A;
  synonym pred_symbol_1 B for pred_symbol_1 A,B;
end;
```

定義の場合のように、ここでも局所変数の繰り返し、あるいは使用されない局所変数は許されないことに注意してください。

モードの場合、同義語は非拡張モードとしてのみ使用可能です。　ですから以下に続く表記は正しくありません：

```
notation
  let X be set;
  synonym mode_symbol of X for Subset of X;
::>                                  *134
end;
::> 134: Cannot redefine expandable mode
```

なぜなら、**Subset of X** は拡張可能モードとして定義されているからです（**Element of**

bool X）。

Mizarの同義語の重要な特徴はオリジナルのコンストラクタの再定義された型を継承しないことです（再定義についてのそれ以上の情報は Section 2.4 を見て下さい）。 例文を見てみましょう、ここで **Morphism of a,b** は同一のカテゴリー（圏）のオブジェクトの間の恣意的なモーフィズム（射）を表します。 次の例の中で型は母型 **set** について定義されます：

```
definition
  let C be non void non empty CatStr, a,b be Object of C;
  mode Morphism of a,b -> set means
  ...
end;
```

そして型は **C** のモーフィズム（基礎となる圏の全ての射）として再定義されます。

```
definition
  let C be non void non empty CatStr, a,b be Object of C;
  redefine mode Morphism of a,b -> Morphism of C;
  ...
end;
```

同義語を考えると：

```
notation
  let C be non void non empty CatStr, a,b be Object of C;
  synonym morphism of C,a,b for Morphism of a,b;
end;
```

それから以下の証明の抜粋に見られるように、同義語は **Morphism of C** の型を保持していません：

```
set C = the non void non empty CatStr;
set a = the Object of C;
set b = the Object of C;
set m = the morphism of C,a,b;
m is Morphism of a,b;
m is Morphism of C;
::>              *4
::> 4: This inference is not accepted
```

ファンクタの同義語の導入にも類似の規則が適用されます。　述語と形容詞の場合、反意語を導入することができます。　反意語の導入に関する規則は同一です。　ここではそれらの使用の一つの例を提示するだけにしましょう。

```
notation
  let i be Integer;
  antonym i is odd for i is even;
end;
```

それから **non even Integer** のような人工的に見えるフレーズを使わないで、より自然な記述を使います。

[7] の論文は定義を有効にする有用なヒントを教えてくれます。

2.4 再定義　（Redefinitions）

その変更が場合によってはより特異的な（狭い）引数で証明可能であるならばあるコンストラクタのデフィニエンスあるいは型を変更するのに再定義が使われます（構造体定義は再定義できないことに注意してください）。　再定義されたコンストラクタと再定義された部分の性質（kind）に依存しますが、個々の再定義は新しい定義が古いものと適合する（compatible）ことを保証する、対応する正当性条件（訳注：の証明を）を誘発します。詳細な記述は Appendix A にあります。

最も一般的な等式（同値）の概念（notion）を考えてみましょう：

```
definition
  let x, y be set;
  pred x = y;
end;
```

述語は特定された（狭い）引数の文脈でより良く記述されます。　例えば等式の引数が関係だと判っているとします、するとそれらが等しいという事実は対の概念（notation of pairs）を使って表現できます。

```
definition
  let P, R be Relation;
  redefine pred P = R means
  for a,b being set holds [a,b] in P iff [a,b] in R;
end;
```

同様に、関数の等式はそれらの関数を共通の定義域に適用して結果を較べた時最も良く表現できます。

```
definition
  let f, g be Function;
  redefine pred f = g means
  dom f = dom g & for x being set st x in dom f holds f.x = g.x;
end;
```

特性がオリジナルの（もっと一般的な）述語を保持していない時に特性（properties）を導入するためにも再定義が使われます（Section 2.5 を見てください）。 例えば、以下の一般演算（generic operation）のように：

```
definition
  let M be multMagma;
  let x, y be Element of M;
  func x*y -> Element of M equals
  (the multF of M).(x,y);
end;
```

結果は可換（commutative）である必要はありません。 しかし、もし基盤となっている構造体が適切であるなら（再定義の中で可換であるなら）特性は以下のように記述できます：

```
definition
  let M be commutative non empty multMagma;
  let x, y be Element of M;
  redefine func x*y;
  commutativity;
end;
```

ここでMizarは再定義の *flat* concept を使うこと、すなわち再定義はいつでも最初のオリジナルのコンストラクタを再定義し、決して導入された再定義を再、再定義しないことを注意しておかないとなりません。

もうひとつの重要な制限は、本来のパターン（proper pattern）だけが再定義で使用できるということです。 言い換えると局所変数に定数を代入したコンストラクタの再定義はできません。 例えば、以下の定義のように：

```
definition
  let X, Y be set;
  func X \/ Y -> set means
  x in it iff x in X or x in Y;
```

```
end;
```

局所変数に代入された定数項 **NAT** を持つ再定義は許されません。

```
definition
  let X be set;
  redefine func X \/ NAT -> non empty set;
::>                  *113
  coherence;
end;
::> 113: Unknown functor
```

コンストラクタの型を変える再定義は、新しい型がオリジナルの型にまで拡張する時（widens to：自動詞）だけに使用されます。　下記の定義を考えてみましょう：

```
definition
  let f be Function;
  assume f is one-to-one;
  func f" -> Function equals
  f~;
end;
```

それから、有効な再定義の例は引数が順列（permutation）の場合のものでしょう、そして新しい結果の型は：

```
definition
  let X be set;
  let f be Permutation of X;
  redefine func f" -> Permutation of X;
end;
```

となります。　それゆえオリジナルの型を制限することは正しくありません、そして次の再定義の結果はエラーになります。

```
definition
  let X be set;
  let f be Permutation of X;
  redefine func f" -> set;
::>                    *117
  coherence;
```

```
end;
::> 117: Invalid specification
```

再定義に使われる局所変数の順番はオリジナルの定義と一致する（match）ことが要請されるのに注意してください。　例えば、定義：

```
definition
  let f be Function, x be set;
  func f.x -> set means
  [x,it] in f if x in dom f otherwise it = {};
end;
```

について以下に示すように、もっと特定された（狭い）文脈で再定義しようと思う人がいるかもしれません、しかしエラーがでます。

```
definition
  let C be non empty set, D be set;
  let c be Element of C;
  let f be Function of C,D;
redefine func f.c -> Element of D;
::>            *109
end;
::> 109: Invalid order of arguments of redefined constructor
```

この場合、解決法は以下のように **f** と **c** の順番を入れ替えることでしょう：

```
definition
  let C be non empty set, D be set;
  let f be Function of C,D;
  let c be Element of C;
  redefine func f.c -> Element of D;
end;
```

上にあげた例は、オリジナルと較べて余分の局所変数（extra loci）が追加されていることを示しています。　現行のMizarの実装では、たとえそれが使用可能な（より少ない）引数の型を定義の型まで拡張する場合であっても、局所変数の数を減らすことは許されていません。

2.5　特性（Properties）。

Section 2.3 で述べたように定義された概念のあるものは、その定義ブロックの中でステー

トメントと*特性*の証明（proofs of specific *properties*）を伴うことができます。 これらの特性は定義と一緒に格納され、MIZAR CHECKER は概念を使う時はいつでも自動的に対応する公式を追加の発見的問題解決法（extra heuristic）として使うことができます。 特性のあるものは再定義（Section 4.2 を見てください）で導入され、そこでは局所変数の型はもっと特異的（狭い）でしょう。

2.5.1 *射影性（Projectivity）*。 これはその関数を２回適用しても１回適用の場合と結果が変わらない、すなわち **f(f(x)) = f(x)** と記述される関数の特性です。 以下はこの特性を持つ関数定義の典型的な例です：

```
definition
  let x be real number;
  func sgn x -> integer number equals
  1 if 0 < x, -1 if x < 0 otherwise 0;
  ...
  projectivity;
end;
```

射影性の特性（**projectivity** property）は一つの可視引数を持つ関数にのみ適用されます、そして結果の型はかならず引数の型まで拡張されます。 もし特性が規定されているなら（stated）、対応する正当性条件がその定義のなかで証明されていないといけません（詳細はAppendix A を見てください）。 現行の Mizarの実装では、この特性は再定義の中では使用できません。

2.5.2 *退縮性（Involutiveness）*。 これはその関数を２回適用すると結果がもとの引数と同じになる、すなわち **f(f(x)) = x** と記述される関数の特性です。 典型的な例は逆元（inverse element）を求める演算でしょう、例えば群の（定義の）中で：

```
definition
  let G be Group, h be Element of G;
  func h" -> Element of G means
  h * it = 1_G & it * h = 1_G;
  ...
  involutiveness;
end;
```

ここで、それもまた可視引数が１つのときだけ意味があります。 結果の型は引数の型と同じであることが要求されます。 **退縮性（involutiveness）**を持つ定義で証明すべき正当条件は Appendix A にあります。 現行Mizarの実装では、再定義の中ではこの特性は

使用できません。

2.5.3 *冪等性（Idempotence 訳注；ある操作を１回行っても数回行っても結果が同じという性質）。* ２つの同じ値に適用された冪等の２項演算は結果としてその値をかえします。 例えば、集合論のユニオン（合併集合）はこの特性を持ちます（**x \/ x = x**）：

```
definition
  let X,Y be set;
  func X \/ Y -> set means
  x in it iff x in X or x in Y;
  ...
  idempotence;
end;
```

冪等性（idempotence）という特性は２つの同じ型の可視引数を持つファンクタに適用されます。 **冪等性**を持つ定義で証明すべき正当性条件は Appendix A にあります。 現行のMizarの実装では、この特性は再定義の中では使用できません。

2.5.4 *可換性 (commutativity)。* これは単純に引数の交換を許す２項演算の特性です。もう一度言いましょう、集合論のユニオン（合併集合）は明らかにこの特性（**x \/ y = y \/ x**）を持ちます。

```
definition
  let X,Y be set;
  func X \/ Y -> set means
  x in it iff x in X or x in Y;
  ...
  commutativity;
end;
```

可換性（commutativity）という特性は２つの同一の型の可視引数を持つファンクタに適用されます。 要請される正当性条件についてのこれ以上の情報は Appendix A を見てください。 **可換性**は再定義の中でも使えることに注意しましょう。

サポートされる５種類の特性で２項述語に適用できるものを以下に提示します（それらは定義と再定義の中で導入できるものです）。

2.5.5 *反射性（Reflexivity）。* 【訳注：reflexivity に「再帰性」を割り当てると社会学の reflexivity（cf. Anthony Giddens）を強く想起させるので、ここでは「反射性」を用いました】

どの引数でもそれ自身がその関係式の中にあるとき、その関係は反射的であるといわれます。　例として被整除性（divisibility）の関係を取り上げましょう。

```
definition
  let i1,i2 be Integer;
  pred i1 divides i2 means
  ex i3 being Integer st i2 = i1 * i3;
  reflexivity;
end;
```

正当性条件として証明される必要のある式の記述については Appendix A を見てください。

2.5.6 *非反射性（Irreflexivity）*。　逆にどの引数も、ちょうど例にある真部分集合の包含関係（proper inclusion）の場合のように、それ自身が決してその関係式の中に存在しない時、この関係は非反射性とよばれます。

```
definition
  let X,Y be set;
  pred X c< Y means
  X c= Y & X <> Y;
  irreflexivity;
end;
```

Appendix A に正当性条件として証明が要請される式を提示します。

2.5.7 *対称性（Symmetry）*。　対称関係においては引数の交換ができます。　明らかに対称的な述語を下に提示します。

```
definition
  let X,Y be set;
  pred X misses Y means
  X /\ Y = {};
  symmetry;
end;
```

正当性条件として証明が要請される式については Appendix A を見てください。

2.5.8 *非対称性（Asymmetry）*。　非対称的な述語とは引数の交換をすると必ずその論理値が変化するもののことです。　真部分集合の包含は明らかに非対称関係なので**非反射性（irreflexivity）**を提示するために使用した一例をここでも使用することができます。

```
definition
  let X,Y be set;
  pred X c< Y means
  X c= Y & X <> Y;
asymmetry;
end;
```

対応する正当性条件として証明が要請される式を Appendix A に提示します。

2.5.9 *連結性（Connectedness）*。 どんな引数 **a** と **b** に対しても **(a,b)** あるいは **(b,a)** がその関係のメンバーであるとき、その２項関係は連結であるといいます。 典型的な例は順序数について定義された包含関係です。 対応する正当性条件として証明される必要がある式のこれ以上の情報については Appendix A をみてください。

```
definition
  let A,B be Ordinal;
  redefine pred A c= B means
  for C st C in A holds C in B;
connectedness;
end;
```

2.6 登録 （Registrations）

ありふれた名前（common name）である*登録（registration）*は、Mizar言語では形容詞を基礎とした型情報の自動処理に関連した、Mizarの特徴的機能（feature）のことです。 いわゆる*クラスター（clusters）*のなかの形容詞のグルーピング（それゆえ、クラスター（房、塊）がそれらの記法として使われます）はある型を推論する規則の自動化を実現可能にします（このメカニズムの詳細にわたる記述は文献 **[17]** を参照）。

最初の種類はMizarの型が空でないことを担保するために使われる*存在に関する*登録（*existential* registration）です。 例えば次の登録は有限でかつ空でない集合が少なくとも１つ存在するという証明（**存在**正当性条件、**existence** correctness condition と言われます）を与えます。

```
registration
  cluster finite non empty set;
  existence
  ...
end
```

もうひとつの種類の登録は*条件付き*登録（*conditional* registration）と呼ばれます。 す

べての空集合はまた有限でなければならない、と記述してある例でその記法（syntax）を示しましょう。

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

ここでMizar記法は矢印 **->** の前の形容詞のリストが空であるのを許すことに注意しましょう。　この場合登録は与えられた型に使われる（associated with）形容詞を記述するのに使われます。　例えば：

```
registration
  let X be finite set;
  cluster -> finite Subset of X;
  coherence
  ...
end;
```

条件付き登録（conditional registration）として記述された形容詞の依存性はMizar verifierが自動的に使用します。　しかし、その（訳注：Mizar verifierの）処理過程をANALYZER モジュールの中でのそれと CHECKER モジュールの中でのそれとに区別することは重要です。　CHECKER の中で行われる計算（Section 3.5 を見てください）は登録のある種の結果（some consequences）を、例えば条件付き登録の対立（contraposition）の推測（infer）を可能にします。　これは ANALYZER の中では不可能です、ある場合にはこのモジュール（CHECKER）の中で行われる型のチェックは：

```
cluster infinite -> non empty set;
```

と、形容詞の因果関係の結果が CHECKER にとって明らかであったとしても明示的に記述された登録が必要になることがあります（**infinite** は **finite** の反意語として定義されています）。

ユーザは存在性の登録は、使用できる条件の帰結とともに拡張される（extended）という事実に気が付いていないといけません。　例えば、使用可能な次の２つの条件付き登録：

```
cluster empty -> Relation-like set;
cluster empty -> Function-like set;
```

と、それに続く存在性の登録：

```
cluster empty set;
```

は実際に次の型をかき集め（*rounded up*）の形容詞クラスタ：

```
cluster empty Relation-like Function-like set;
```

として登録します。

データベースからの登録のインポートが成功するのは、アーティクルの環境部でその形容詞のすべてのコンストラクタが使用可能な場合に限られているので、これらの特徴的機能を知ることが極めて重要になります（与えられた環境部において特定の登録に必要なすべてのコンストラクタを検出するために CONSTR ユーティリティを使うこともできます、Section 4.5 を見てください）。

最後の３番目の種類の登録は*項形容詞登録（term adjectives registrations）*でMizarのジャーゴンでは時々*関数記号登録（functorial registrations）*と呼ばれることもあります。名前が示唆するように、それらは指定された項により保持されるある特性（property）を登録するのに使用されます。　再定義とは対照的にパターンを使えるだけでなく、もっと複雑な項も使用できます。　項はファンクタ、セレクタ、凝集体（aggregate）あるいは忘れっぽいファンクタ（forgetful functor）を使って構築されます。
以下は例です：

```
registration
  let X be non empty set;
  let Y be set;
  cluster X \/ Y -> non empty;
  coherence
  ...
end;
```

（余分な型を持つ）項形容詞登録（term adjectives registrations）の拡張記法（extended syntax）は、再定義型を持つ項だけに適用可能な形容詞の登録に使うことができます。　次の登録の例を考えてみましょう。

```
registration
  let T be TopSpace;
  let X, Y be open Subset of T;
  cluster X \/ Y -> open Subset of T;
  coherence
  ...
end;
```

この場合、形容詞 **open** は項 **(x \/ Y)** の型が再定義された変異形（variant）（**T** の部分集合）であると適切に識別された（properly identified）場合のみ正しいのです。　余分な型の変更を伴う登録が、再定義がやるようにすでに登録された項の型を変更するというのは（類似した記法の結果としての）ありがちな誤解だということを心にとめておいて下さい。　その反対に、そういう登録を受け入れるために、システムは与えられた項が（それぞれ対応する再定義の結果として）本来の提供された型であることを推測できなければなりません。

最後にここで一般的な法則として、通常、登録は再定義より好ましいと述べておきましょう。　第一の理由は登録が新しいコンストラクタを導入しないからです。　さらに登録経由でインポートされた形容詞クラスタは単純に集積された（cumulated）ものです、いっぽう再定義の*フラット*の概念（*flat* concept）は使用可能な最後の再定義だけにしかアクセスできません。　これ以上のMizarにおける形容詞の内部的な処理についての情報は文献 **[11]** で見ることができます。

2.7 項の識別　（Terms identification）

数学の現場では与えられたオブジェクトあるいは演算は、問題が発生した文脈に依存してしばしば非常にたくさんの異なった方法で取り扱われます。　自然数はフォン・ノイマン数、あるいは有限の順序数（a finite ordinal）と考えることもできます。　最小公倍数は数の演算、あるいはある束の要素の最小上界（supremum of element of some lattice）とみなすことができます、等々。　その場合、自動的「翻訳」定理（automatic 'translation' theorem）をもつことは、しばしばとても有用です（worthwhile）。　Mizarではこれは **identify** キーワードを使った*項識別（term identification）*によって行われます（技術的には登録のブロックに実装されています）。　例えば：

```
registration
  let p, q be Element of Nat_Lattice;
  identify p "\/" q with p lcm q;
  compatibility;
end;
```

この識別の狙いは 、**with** キーワードの両側に項が一緒に出現した時にはいつでも行われる、左側の項と右側に記述された項のマッチングです。　現行の実装では、一方向のマッチングだけが許されています、すなわちベリファイヤが左側の項を含む文を処理しているときにそれ（ベリファイヤ）は左側の項のシンボルを右側のもので置き換えたローカルコピーを生成し両方の項がお互いに等価になるようにします。　その等式は右側のものについて書かれた補題を経由して左側の項についての事例（fact）をジャスティファイすることを許します、しかし逆は真ではありません（not vice versa）。　この意味において識別は対称的ではありません、それを **Nat_Lattice** は自然数の束で最大公倍数（**lcm**: least

common multiple）と最大公約数（**hcf**: highest common factor）の（計算）を演算として持つ、次の例で示します。　そこで次の補題：

```
L1: for x, y, z being natural number holds
    x lcm y lcm z = x lcm (y lcm z);
```

が文

```
L2: for x, y, z being Element of Nat_Lattice holds
    x "\/" y "\/" z = x "\/" (y "\/" z) by L1;
```

をジャスティファイするのに使うことができます。　しかし **L1** を **L2** で直接ジャティファイするのはうまくいきません。

項の識別（terms identification）は導入された場所の直後からアーティクルの最後まで使用可能です。　もし外部のアーティクルで導入された識別を使用したければ、環境部でインポートされている必要があります。　現行の識別は内部的には登録と類似の実装をされています、なので識別は自分自身についてのライブラリ指令を持っていません、そこで識別は**登録**と一緒にインポートされます。

下に記すのは項の識別が作動する時に出力される典型的なエラーです：

```
registration
  let p, q be Element of Nat_Lattice;
  identify p "\/" q with 1_NN;
::>                          *189,189
end;

::> 189: Left and right pattern must have the same number of arguments.
```

この場合 CHECKER が提供するエラーコードは自己説明的（self-explanatory）です。

```
registration
  let p, q be Element of Nat_Lattice;
  let m, n be Nat;
  identify p "\/" q with m lcm n when p = m, q = n;
::>                                      *139    *139
end;

::> 139: Invalid type of an argument
```

このエラーは変数 **p** と **q** の型が、それぞれ **m** と **n** の型にかき集められない（round up to）ことを意味しています。　この問題の解決法は登録を使うことです。

```
registration
  cluster -> natural Element of Nat_Lattice;
  coherence;
end;
```

2.8 定義、再定義と登録のまとめ （Summary of definitions, redefinitions and registrations）

下のテーブルは特定の種類の定義に適用できる、要求される正当性条件と特性の使用法をまとめたものです。

Definitions		
	Correctness conditions	Properties
Predicate	—	reflexivity irreflexivity symmetry asymmetry connectedness
Attribute	—	—
Mode	Existence	—
Functor	existence uniqueness	commutativity idempotence involutiveness projectivity

再定義の場合は、要請される正当性条件と特性は結果の型あるいはデフィニエンスが変更されるかどうかに依存します。

Redefinitions changing the result type		
	Correctness conditions	Properties
Mode	coherence	
Functor	coherence	commutativity
Redefinitions changing the definiens		
	Correctness conditions	Properties
Predicate	compatibility	reflexivity irreflexivity symmetry asymmetry connectedness
Attribute	compatibility	
Mode	compatibility	
Functor	compatibility	commutativity

下に、特定の種類の登録をジャスティファイするために必要な正当条件のリストを提示します。

Registrations	
	Correctness conditions
Existential	existence
Conditional	coherence
Functional	coherence
Term identification	compatibility

部分定義（partial definition）は追加の**整合性**条件（**consistency** condition）を要求することを見てください（Appendix A を見てください）。 さらに、全称正当条件（universal correctness condition）は、与えられた定義あるいは登録で証明されるべく残っている全ての正当性条件の置き換えとして **correctness** とタイプすることができることに留意してください。

3. システム（SYSTEM）

Mizarのベリファイヤは Mizarアーティクルのいろいろな面の正当性をチェックすることに責任がある数個のモジュールから構成されます。 それらは: SCANNER, PARSER, ANALYZER, REASONER, CHECKER, SCHEMATIZERです。

3.1 スキャナ-トークナイザ（Scanner-tokenizer）

スキャナはソース・ファイルを読んでそれをトークンに切りだします。 最初のステップとしてすべてのコメント（ダブルコロン**::**で始まる行の部分）が除去されます。 次にモジュールはホワイトスペースで区切られたテキストのセグメントを分析し、トークン －カテゴリーの一つに分類できる、可能な限り最も長い文字列－ に切りだします：

予約語（reserved words） － 全リストが Section 2 にあります。

シンボル（symbols）－ ユーザのボキャブラリ・ファイルで定義された概念を示すために導入されます； シンボルは32より大きい（訳注：aboveなので>32）7ビットASCIIコードです。

数（numerals） － 0でない数字から始まる数字の一続き（というわけで（訳注：最初の）0 は数としてあつかわれません）； 数の数値は32767を超えることはできません（符号付き16ビット整数の最大値）

識別子（identifiers）－ 予約語、シンボル、数ではない文字または数字のストリング；

識別子は変数、スキーム、そしてラベルを命名するのに使われます； その規則に従えば（テキスト）**01** は有効な識別子です。

ファイル名（filenames）－ 8 文字までの大文字のストリング（アルファ・ニューメリックおよびアンダースコア）； 環境部の指令では選択したアーティクルから、そしてアーティクルの主要な部分ではライブラリ・リファランスからアイテムをインポートするために使われます。

最後に、ここで大文字／小文字はMizarでは意味があることを思い出しましょう。

3.2 パーサ（Parser）

MizarのPARSER の主な役割は文法（grammar）に関して、SCANNER が生成したトークンのストリームが記法上（syntactic）正しいことをチェックすること（Appendix B を見て下さい）、および ANALYZER が使用するための、スタックの形にしたブロックとアイテムをアーティクルのアブストラクト表記として作成することです。

この言語の文法規則で規定されていない、ある種の特徴を考慮に入れた表記（representation）が作成されるのに注意するべきでしょう： それらの特徴とはインポートされた演算（operation）の優先度（priority）と引数の数（arity）、そして長い項の分析のために特別に作成されたアルゴリズムです(詳細な記述は文献 **[19]** を見てください)。カッコの数を減らすために追加の規則が連想の概念（associative notation）を論理積（conjunctive）と論理和（disjunctive）の公式に対して使用できるように指示（specify）します、しかし含意と同値の公式については指示しません。 さらに、論理接続詞（logical connectives）の結合力は量化子のそれよりも強力です。

3.3 アナライザ（Analyzer）

ANALYZER の主な役割は使用されたコンストラクタと概念（notations）を利用可能な登録と環境部からインポートされた型情報に基づいて識別（identification）する（曖昧さをなくす（disambiguation））ことです。

もし（シンボルのオーバーロードの結果として）複数のやり方の識別が可能ならば、**notation** 指令の中の最後ものが有効です。 例えば、同じ＋のシンボルで表記される2つの加算の演算が、一つは複素数（complex numbers）のためのもので、他のものは拡張された（±∞を持つ）実数（extended real numbers）のためのものだとします。
それでシステムが数 1 と数 2 が２つの型の両方を持つことを推測（infer）できるという条件で、1+2 という表現（expression）には２つの解釈がありえます。 ANALYZER は利用可能な概念（notation）のリストの最後にある演算を選択するでしょう。

しかし、Mizarでは強制的に選択された識別を使用するメカニズムが利用可能です。 上の例に関して：

```
1 qua complex number + 2
```

というフレーズは数 1 を複素数として取り扱うように ANLYZER に強制します、その結果（訳注：シンボル＋の）拡張された実数に対する変異型（variant）は使用されません。

3.4 リーゾナ（Reasoner）

REASONER モジュールは（訳注：アーティクルの）著者が使う証明の戦術が証明中の公式に対応しているときは照合（checking）に責任があります。 照合は公式の内部的な表示（representation）であるそれらの単純化した「正準」形（“canonical” form） － **VERUM, not, &** そして **for _ holds _** と原子式（atomic formulas）だけを使用する「意味論的相互関連」（semantic correlates）に基づいて行われます。 その他の公式は以下の規則のセットを使ってエンコードされます。

— **VERUM** は論理積の中性要素です（neutral element of the conjunction）；
— 二重否定の規則が使われます；
— ド・モルガンの法則は論理和（disjunction）と存在量化子で使われます；
— α **implies** β は **not(**α **&** **not** β**)** に変換されます；
— α **iff** β は α **implies** β **&** β **implies** α に変換されます、すなわち **not(**α **&** **not** β**) &** **not (**β **&** **not** α**)** に変換されます；
— 論理積（conjunction）は結合律を満たす（associative）が、可換（commutative）ではありません。

それらの単純化のおかげで、それ（Reasoner）が生成する semantic correlate が証明中のステートメントのそれ（semantic correlate）と同一である限り証明の骨格は有効であると考えられています。

3.5 チェッカー（Checker）

Mizarシステムの最も複雑なモジュールは CHECKER で、古典的な反証装置（classical disprover）として作動します。 最も重要なことですが、そのモジュールの中で

$$\frac{\alpha^1, \ldots, \alpha^k}{\beta}$$

の形の推測（inference）は

$$\frac{\alpha^1, \ldots, \alpha^k, \neg\beta}{\bot}$$

という形に変換されます。 （訳注：記号⊥は論理記号で空節（empty clause）を表し、空節は偽を意味する*） それから、前提条件（premiseS）の論理和正準形（disjunctive normal form, DNF）が作成されシステムはそれに反証（refute）しようと試みます。

---

*萩谷昌巳、西崎真也 著 「論理と計算のしくみ」岩波書店、２００７年 ４０ページ参照

$$\frac{\left([\neg]\alpha^{1,1}\wedge\ldots\wedge[\neg]\alpha^{1,k1}\right)\vee\ldots\vee([\neg]\alpha^{n,1}\wedge\ldots\wedge[\neg]\alpha^{n,kn})}{\bot}$$

ここで $\alpha^{i,j}$ は、原子文あるいは（否定文あるいは否定文でない）全称文（atomic or universal sentences）です － 受けいれられる予定の推測（inference）について、全ての論理和（all disjuncts）は反証されねばなりません。 そこでn 推測は（n inference）は実際にチェックされます。

$$\frac{\left([\neg]\alpha^{1,1}\wedge\ldots\wedge[\neg]\alpha^{1,k1}\right)}{\bot}$$

. . .

$$\frac{([\neg]\alpha^{n,1}\wedge\ldots\wedge[\neg]\alpha^{n,kn})}{\bot}$$

反証の過程で CHECKER は*明白な推測（obvious inference）*の概念と一体化した数多くの発見的問題解決法（heuristics）を使います。 処理（processing）は全ての項、使用されたコンストラクタの特性、等式計算（equality calculus）、同等拡張（equals expansion）、同定登録（**identify** registrations）に伴う型情報に関係（concern）します。 また CHECKER は特別の組み込みの自動操作手続きを、例えば複素数（直接計算）や集合のブール演算のような一部の頻繁に使用されるオブジェクトを処理するために使用します。 これらの内部ルーチンは requirement 指令を使うことで実行されますが、詳細は文献 [13] に記述されています。

3.5.1 *スキマタイザ（Schematizer）*。 Mizarは標準的数学で頻回に使われる１階述語論理を超えたステートメントのエンコードを実現可能にするため、いわゆる*スキーム（schemes）*をサポートします。 Mizarでは定理のスキーム（定理の無限族）を形成するために自由２階変数が使えます。 その記法は典型的な例、例えば帰納法のスキームで最も良く見ることができます：

```
scheme :: NAT_1:sch 2
  NatInd { P[Nat] } : for k being Nat holds P[k]
provided
  P[0] and
  for k being Nat st P[k] holds P[k + 1];
```

２階変数が述語を表しているなら、それは後に角カッコの中にある引数の型のリストをもつ識別子で記述されています（例： **P[Nat]**）。 ファンクタの場合には単純なカッコが使われ結果の型が記述されます（例： **F(Nat) -> Nat**）。 空の引数リストをもつファ

ンクタはスキーム*定数（scheme constants）*と呼ばれます。

局所的には（スキームを導入したアーティクルの中では）スキームはそのラベルで参照されます（この例では **NatInd**）。 Mizarのデータベースに格納されているスキームはライブラリ参照により参照できます。 スキームを使うためには以下に示す３つのステップに従う必要があります。

（1） スキームの記述に従って、スキームの変数のパターンと一致したプライベートのファンクタあるいは述語を定義し

（2） スキームの前提を証明し

（3） ステートメントのジャスティフィケーションのなかで、**from** キーワードを使ってスキームへの参照を行う（外部スキームは **schemes** 環境部指令によりインポートされていなければならないし、前提の順番はスキームでの宣言のそれと正確に一致していないといけません）

この３つのステップに対応する（**::step 1** から **::step 3** の）コメントをつけたスキームの使用法の例を以下に提示します。

```
2 divides n * (n+1)
proof

  :: step 1 - private predicate
  defpred P[Nat] means 2 divides $1 * ($1+1);

  :: step 2 - premises
a1: P[0];
a2: for k being Nat st P[k] holds P[k+1];

  :: step 3 - referring to the scheme
  for k being Nat holds P[k] from NAT_1:sch 2(a1,a2);
  hence 2 divides n * (n+1);
end;
```

以下にスキームを使う場合に出会う典型的なエラーを示します。

スキーム：

```
scheme :: NAT_1:sch 12
  { D() -> non empty set, A() -> Element of D(),
    G(set,set) -> Element of D() }:
  ex f being Function of NAT,D() st f.0 = A() &
```

```
   for n being Nat holds f.(n+1) = G(n,f.n);
```

があって、再帰関数（recursive function）の存在の証明にそれを使いたいと考えているとします、例えば：

```
definition
  let s be Real_Sequence;
  func Partial_Sums(s) -> Real_Sequence means
:: SERIES_1:def 1
  it.0 = s.0 & for n being Nat holds it.(n+1) = it.n + s.(n+1);
end;
```

正しい使用法は以下に示すようになるでしょう。

```
deffunc U(Nat,Real) = $2 + s.($1+1);
consider f being Function of NAT,REAL such that
f.0 = s.0 & for n being Nat holds f.(n+1) = U(n,f.n)
from NAT_1:sch 12;
```

ここでスキーム定数 **D()** と **A()** には **REAL** と **s.0** が、そしてスキーム変数 **G(set,set)** にはローカル・ファンクタ **U** が、それぞれ代入されます。 これは置換する項の型がスキームの宣言に合うのでうまくゆきます、すなわち **REAL** は **non-empty set** 型で、**s.0** は **Element of REAL** 型で **U** の結果も同様に **Element of REAL** 型です。

もし **f** の代わりに無関係な **g** を使用しようとするとエラーが報告されるでしょう：

```
consider f being Function of NAT,REAL such that
g.0 = s.0 & for n being Nat holds f.(n+1) = U(n,f.n) from NAT_1:sch 12;
::>                                                    *20
::> 20: The structure of the sentences disagrees with the scheme
```

同様に **s.0** の代わりに、期待されている型（この例では **Element of REAL**）と異なる他の型を持つ別の項、例えば（say）**s** が使われると：

```
consider f being Function of NAT,REAL such that
f.0 = s & for n being Nat holds f.(n+1) = U(n,f.n) from NAT_1:sch 12;
::>                                                    *26
::> 26: Substituted constant does not expand properly
```

とエラーが発生します。 もう一度言いましょう、例えば（本来の持つべき型を持たない）**s** を **$2 + s.($1+1)** の代わりに **deffunc** 定義のなかで使用することもやはり無効で

す：

```
deffunc U(Nat,Real) = s;
consider f being Function of NAT,REAL such that
f.0 = s.0 &
for n being Nat holds f.(n+1) = U(n,f.n) from NAT_1:sch 12;
::>                                              *30
::> 30: Invalid type of the instantiated functor
```

前に述べたようにスキームを使うためには、はじめにスキームの変数とパターンが一致したプライベート・ファンクタあるいは述語を定義しなければなりません。　しかし、スキームの述語あるいはファンクタと正確に同じ数と順番の引数を持つ適合した式あるいは項が代入される場合はこのステップは不必要です。　**f(n) = n!** となる関数 $f : \mathbb{N} \to \mathbb{N}$ を構成（construct）しようとしている次の例を分析してみましょう。　以下のスキームを使うことができます：

```
scheme :: FUNCT_1:sch 4
  Lambda { D() -> non empty set, F(set) -> set } :
    ex f being Function st dom f = D() &
      for d being Element of D() holds f.d = F(d);
```

D() に自然数の集合 **NAT** を代入し、**F(set)** にローカル・ファンクタを代入して

```
Deffunc F(Element of NAT) = $1!;
```

以下のように関数 **f** を得るため

```
ex f being Function st dom f = NAT &
for d being Element of NAT holds f.d = F(d) from Lambda;
```

しかし **F(set)** と **!** の両方とも単項のファンクタ（unary functors）なので、同じ結果が直接に：

```
ex f being Function st dom f = NAT &
for d being Element of NAT holds f.d = d! from Lambda;
```

と得られます。　しかし、もし **f(n) = 2*n** であるような関数 $f : \mathbb{N} \to \mathbb{N}$ を導入しようとするとこの方法はうまくいきません。

```
ex f being Function st dom f = NAT &
for d being Element of NAT holds f.d = 2*d from Lambda;
::>                                              *28
```

```
::> 28: Invalid list of arguments of a functor
```

このエラーは **F(set)** が引数を一つだけ持つことを期待されているのに、乗算（*****）が２項演算だからです。　この場合このスキームを使用するにはプライベート・ファンクタの定義が必要です。

```
deffunc G(Element of NAT) = 2*$1;
ex f being Function st dom f = NAT &
for d being Element of NAT holds f.d = G(d) from Lambda;
```

プライベートな定義は引数の順番が正確に一致していないときにも必要です。　例えば $f(m,n) = m^n$ であるような関数 $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ を作るために次のスキーム：

```
scheme :: BINOP_1:sch 4
  Lambda2D { X,Y,Z() -> non empty set,
             F(Element of X(),Element of Y()) -> Element of Z() } :
  ex f being Function of [:X(),Y():],Z() st
  for x being Element of X() for y being Element of Y()
    holds f.(x,y) = F(x,y);
```

を使うこともできます。　以下のようになります。

```
ex f being Function of [:NAT,NAT:],NAT st
for x,y being Element of NAT holds f.(x,y) = x |^ y from Lambda2D;
```

しかし、$f(m,n) = n^m$ が欲しい時には、エラーが報告されます。

```
ex f being Function of [:NAT,NAT:],NAT st
for x,y being Element of NAT holds f.(x,y) = y |^ x from Lambda2D;
::>                                                    *28
::> 28: Invalid list of arguments of a functor
```

なぜなら **x** と **y** の順番が逆だからです。　この状況でスキームを使うには：

```
 deffunc F(Element of NAT,Element of NAT) = $2 |^ $1;
```

のようなプライベート・ファンクタが次のように適用されなければなりません。

```
ex f being Function of [:NAT,NAT:],NAT st
for x,y being Element of NAT holds f.(x,y) = F(x,y) from Lambda2D;
```

同様の制限がスキームの述語の使用にも関係（concern）します。

4. ソフトウェア（SOFTWARE）

4.1 インストレーション（Installation）

MizarシステムとMizar数学ライブラリ（MML）はパブリックでダウンロードができ、誰でも商用目的でなければ料金なしで（free of charge）使用可能です。 Mizarの配布内容の著作権は Association of Mizar Users が保持していて、そこが MML の維持とMizarソフトウェア開発の調整を行います。 システムは多数のオペレーティング・システムで、すぐ使用可能状態のコンパイル済みの形で入手可能です。 最新のMizarのリリース（version7.11.07, MML4.156.1112）は匿名の anonymous FTP あるいは HTTP を介して以下のOS用の物がダウンロードできます。

－Linux (i386)、
－Solaris (i386)、
－FreeBSD (i386)、
－Darwin/Mac OS X (i386)、
－Darwin/Mac OS X (PPC)、
－Linux (PPC)、
－Linux (ARM)、
－Win32

配布物は必要なものを完備している（self-contained）ので、インストールはかなり簡単（straightforward）です。 パッケージは Mizar processor、MML を構成するすべてのアーティクルとそれらのアブストラクト、それらのアーティクルに基づくデータベース、ユーティリティー・プログラムのセット、そして（訳注：Mizar）システムを使うときに便利な GNU emacs の lisp モードを含みます。 インストールには約 220MB のディスク・スペースの空きが必要です。

以下にUnixの類似環境、そして Microsoft Windows 環境でMizarをインストールする方法を示します。

4.1.1 *Unix系のOS（Unix-like OS's）*。

配布物は tar のアーカイブとしてダウンロードできます、例えば **`mizar-7.11.07_156.1112-i386-linux.tar`** です。 アーカイブの名前はMizarシステムのバージョン（ここでは7.11.07）、MML のバージョン（ここでは 4.156.1112）、そしてソフトウェアがその上でコンパイルされたOS（ここでは i386 ベースの Linux）を指示しています。

アーカイブのなかにはスクリプト・ファイル **`install.sh`** があり、これは指定されたディレクトリにMizarシステムをアンパックするのに使われます。 大部分のケースでは引数なしでこれを起動するだけで十分です：

```
./install.sh
```

それからファイルをコピーする場所についての３つの質問に答えます。 新しいバージョンのインストレーションは、以前と同一のディレクトリが指定された時には、バージョン間のコンフリクトを避けるために、いつでも古いバージョンを完全に消去してから置き換えることに注意してください。

このスクリプトとともに使うことができる２つオプションがあります：

```
./install.sh --default
```

はスクリプトをノンインタラクティブ・モードで実行します（デフォルトのディレクトリについては下を見てください）、

```
./install.sh --nodialog
```

はいつでもスクリプトを（インストールのためのセミ・グラフィック・ユーザーインターフェースの対話ユーティリティを使わない）プレーン・モードで実行し、**STDIN** から入力を直接受け取ります（これはシェル・スクリプトの中から使うのに便利です）。

インストレーション・スクリプトを対話モードで実行しているときには、最初にMizarの実行ファイルをインストールしようとしているディレクトリの名前を与えなければなりません(デフォルトは **/usr/local/bin** です)。 このディレクトリがあなたの **PATH** 環境変数の中に含まれていることを確認してください。 それからMizarの共有ファイルのディレクトリの入力を要求するプロンプト（デフォルトは **/usr/local/share/mizar** です）が出現します。

インスールが終わった後、新しい環境変数 **MIZFILES** がこのディレクトリを指すようにセットしなければなりません。 例えば **Bash** シェルを使っているとき、あなたは **Bash** のコンフィグレーション・ファイル **.bashrc** に、この設定をパーマネントにするために以下の行：

```
export MIZFILES=/home/john/my-mizar-files
```

を追加しなければなりません。 そうしないとシステムはデータベースを正しく検索することができないでしょう。

最後にインストレーション・スクリプトがMizarドキュメンテーション・ファイルを置く場所を尋ねます（デフォルトは **/usr/local/doc/mizar** です）。

その後で、**MIZFILES** が正しく設定されているなら、システムは使用準備完了です。 配布物には、システムを便利に使うための GNU emacs の Lisp による初期化ファイルも含まれています。 それを使うにはそのファイル **.emacs** を先ほど選択したMizarドキュメンテーション・ファイルを置く **Mizar doc** サブディレクトリから取り出し、通常はユーザのホーム・ディレクトリに **.emacs** ファイルとして格納されている、あなたの GNU

Emacs の初期化ファイル **.emacs** にアペンドするだけです。

4.1.2 *マイクロソフト・ウィンドゥズ (Microsoft Windows)*。 配布物は自己解凍型の **zip** ファイル、例えば **mizar-7.11.07.4.156.1112-i386-win32.exe** としてダウンロードできます。 アーカイブの名前は配布物にふくまれるMizarシステムのバージョン（ここでは7.11.07）、MML のバージョン（ここでは4.156.1112）を示しています。 このソフトウェアはコンパイル済なので実際上 Microsoft Windows（9x/2000/NT/XP/Vista/7）のどのバージョンでも実行できます。 テンポラリ・ディレクトリで自己解凍ファイルを実行すると多数の圧縮ファイルと、指定されたディレクトリにMizarシステムをアンパックするのにつかわれるスクリプト・ファイル **INSTALL.BAT** ができます。 ほとんどの場合、そのディレクトリにセットされたコマンド・プロンプトからディレクトリの引数を一つ付けて、例えば

```
INSTALL c:¥mizar
```

とそのスクリプトを実行するだけで十分です。 パス **c:¥mizar**は（訳注：Mizar）システムがインストールされるべき他の場所のディレクトリ名とハード・ドライブ文字で置き換えることができます。 パス名が与えられない場合はデフォルトの **c:¥mizar** が使われるでしょう。 新しいバージョンのインストールは同一のディレクトリが指定された時には、バージョン間のコンフリクトを避けるために、いつでも古いバージョンを完全に消去してから置き換えることに注意してください。

インストールが終わった後に新しい環境変数 **MIZFILES** がこの与えられたディレクトリを指すようにセットしなければなりません。 そうしないとシステムはデータベースを正しく検索することができないでしょう。 フル・パスを指定しないでMizarの実行ファイルを走らせることができるように、システム **PATH** 変数にそのディレクトリの名前を追加したほうが良いかもしれません。 比較的新しいバージョンの Windows ではこれらの作業はコントロール・パネル **->** システム **->** アドバンスド・メニューにある環境変数タブの適切なエントリーを編集することで行うことができます。

AUTOEXEC.BAT 構成ファイルを使う、より古いバージョンの Windows システムでは同じ効果を得るために次の行を追加します（必要なら **c:¥mizar** を他のディレクトリ名で置き換えてください）。

```
set PATH=c:¥mizar;%PATH%
set MIZFILES=c:¥mizar
```

その後で、**MIZFILES** が正しく設定されているなら、システムは使用準備完了です。 配布物には、あなたがシステムを便利に使うための GNU emacs の Lisp による初期化ファイルを含みます。 それを使うには、そのファイル **.emacs** を先ほど選択したMizarドキュメンテーション・ファイルを置く **Mizar doc** サブディレクトリから取り出し、あなた

の GNU Emacs 初期化ファイル **.emacs** ファイルにアペンドするだけです。 通常、これはユーザのホーム・ディレクトリ（環境変数 **HOME** か、あるいは **HKCU¥SOFTWARE¥GNU¥Emacs¥HOME** レジストリ・エントリに記述されています）、あるいはメイン・ファイルシステムのルート（**c:¥.emacs**）に **.emacs** ファイルとして格納されています。

4.2 Mizarアーティクルの準備（Preparing a Mizar article）

Mizarアーティクルの準備の技法（mechanics）は次のとおりです：

— ソース・テキストはどの ASCII エディタでも作成でき、典型的なものは1500-5000行からなります。
— テキストは ACCOMMODATOR により実行されます。 環境部の宣言の指令はアーティクルに特異的な環境部の作成を誘導（guide）します。 環境部は使用可能なデータベースから生成されます。
— さて VERIFIER はチェックの開始準備ができています。 出力はソース・テキストの受け付けられなかった部分に所見（remark）がついたものを含みます。

これらの３つのステップはエラーが検出されなくなり、著者が結果のテキストに満足するまでループで繰り返されます。 通常、ACCOMMODATOR と VERIFIER は **mizf**（あるいは**mizf.bat**）ユーザー・スクリプトのなかから呼ばれます。 もう一つの方法として、Josef Urban の Emacs-Lisp Mizarモード（今では全てのMizarの配布物に含まれています）がとても役に立つ（fully functional）Mizarシステムへのインターフェースを提供します。

完了したら、アーティクルは Mizar Mathematical Library への収録（inclusion）をめざして Library Committee of Association of Mizar Users へ投稿されます。 貢献しようとするアーティクルは査読（review）の対象となり、もし必要なら著作者はファイルを改訂しなければなりません。 アクセプトされたアーティクルの内容は EXPORTER と TRANSFER ユーティリティで抽出され全てのMizarユーザに配布されるパブリック・データベースに一体化（incorporated）されます。 もうひとつ別の方法は、ユーザがローカル・データベースを準備するためにコマンド

```
miz2prel article
```

を、そしてファイル名の拡張子が **.abs** のファイルすなわち証明部分を除いた **.miz** ファイルを得るために

```
miz2abs article
```

を実行することです。 EXPORTER により抽出されたデータはカレントディレクトリの **prel** サブディレクトリに格納されます。

4.3 ボキャブラリ（Vocabularies）

ボキャブラリはMizarアーティクルのための追加の用語集（lexicons）を定義します。Mizarアーティクル同様、ボキャブラリ・ファイルも平文（plain）の ASCII ファイルです。　実際にはどのアーティクルからも独立なのですが、その名前はボキャブラリの添付先のアーティクルの名前と同じでなければなりません。　必要な拡張子は **.voc** でそれを **.miz** ファイルといっしょに（訳注： **.miz** ファイルのあるディレクトリの）**dict** サブディレクトリに置かれなければなりません（通常提案される **text** サブディレクトリに置かれなかった場合です）。　ボキャブラリ・ファイルの各行はシンボルを導入します；それはシンボルの種類を決める一文字の修飾子（qualifier）ではじまり、直後にコントロール・コード、スペース、ダブルコロンを含まない恣意的なキャラクタのストリングからなる定義シンボルの表記が続きます。

シンボルの修飾子（qualifiers）は次に示す意味をもちます。

— R - 述語（predicate）、
— O - ファンクタ（functor）、
— M - モード（mode）、
— G - 構造体（structure）、
— U - セレクタ（selector）、
— V - 属性（attribute）、
— K - ファンクタの左カッコ（left functor bracket）、
— L - ファンクタの右カッコ（right functor bracket）。

ファンクタのシンボルに対しては追加の情報、すなわちファンクタの優先度（priority）、がボキャブラリ・ファイルの中に与えられることがあります。　優先度は整数の0から255の間でファンクタの識別子とスペースの後で、その同じ行に書かれます。　指定されていない場合はデフォルトの優先度64ということになります。　例えば加算に対する乗算の慣習的（conventional）優先性（precedence）を実現（facilitate）するために、現行の MML では **+** シンボルはより低い優先度で：

```
O+ 32
```

と導入されている一方、***** シンボルにはデフォルトの値（64）が使われています。

4.4 アコモデータと環境部宣言　（Accommodator and environment declaration）

Mizarアーティクルの環境部宣言は以下に述べる指令（directives）から構成されます：

vocabularies. SCANNER と PARSER が使えるようにリストに記載されたボキャブラリから全てのシンボルをインポートします。

constructors. リストに記載されたアーティクルで定義された全てのコンストラクタ、および、それらを理解するために必要な他のコンストラクタの全てをインポートします。

結果として、もしローカルの環境がコンストラクタを含めば、それはまたその型、およびその引数の型で出現するコンストラクタも含みます。

notations. notation を理解するために必要とされる全てのコンストラクタがすでにインポートされているということであれば、すでにインポートされたコンストラクタによって使われるフォーマットとパターンをインポートします、

registrations. 形容詞、モード、そしてファンクタの間の関係を記述した登録の定義をインポートします。

theorems. リストに記載のアーティクルから定理と定義定理への参照を可能にします。

schemes. 上記とほぼ同様、スキームへのアクセスを与えます。

definitions. 定義の名前を述べないで、定義拡張（definitional expansion）による証明で使用することができる definientia（definiens の複数、定義するものたち）を要請します。 それはまた **equals** で定義されたファンクタの自動的拡張を許します。

requirements. ある種の特別のアーティクルに関連する組み込みの機能（built-in features）にアクセスを与えます。

ACCOMMODATOR(ACCOM) は上記の指令に従ってMizarデータベースから情報を集め、VERIFIER が必要とする補助フィルの束（bunch）を準備します。 もうひとつの別の方法で、Unix の **make** に類似の MAKEENV ユーティリティを使うことができます － それは環境宣言が変更されて中間ファイルがリビルトされるべき時にだけ ACCOM を呼出します。

4.5 補助ユーティリティ（Auxiliary utilities）

使用されたシンボルについての情報（**vocabularies**）はMizar配布資料の中の単一のテキスト・ファイル **mml.vct** に収集されます。 このファイルを使いやすくするためにある簡単な問い合わせツールが付属します。 これらの全ては Emacs のメニューから： Mizar－Voc.Constr.Utilities で実行できます。

FINDVOC. 問い合わせたシンボルを含むボキャブラリを探すこのツールの最も一般的な呼び出し文字列（calling sequence）は

```
findvoc -w <symbol>
```

で、スイッチ **-w** は単語全体の一致を意味します。

どのシンボルについても、その優先度（ファンクタ・シンボルの場合だけです）とその修飾子（qualifier）が与えられます。 優先度（priority）が無い場合はデフォルト（訳注：の64）を意味します。 ボキャブラリ中のシンボルについては、（ボキャブラリとアーティクルの名前空間を分離させてしまうのではなく）そのアーティクルの識別子の初回の使用に合わせる（keep … with … ）というライブラリ委員会のポリシーのため、このユーティリティはどの環境指令に追加の書き込みを加えて延長させるべきかについての大まかな

見方を与えることができます。

LISTVOC.　（ボキャブラリのリストを作成する）このプログラムを MML 識別子 **example** と一緒に呼ぶことで、ボキャブラリ **example** で導入されたすべてのシンボルのリストを得ることができます。　通例（as a rule）、これはそのシンボルを最初に使った（**mml.lar** で与えられる順序を考慮した）アーティクルです、しかしこれは改訂の結果で変わるでしょう。

CHECKVOC.　この（ボキャブラリをチェックする）単純なユーティリティはアーティクルを MML への収録をめざして投稿する前に使うべきです。　それはユーザのボキャブラリが禁止されたキャラクタを含んでいないかをチェックし（Section 4.3 を見て下さい）、１文字のシンボルの使用や他の人が導入したシンボル（すなわちすでに **mml.vct** で使用可能になっている）がないかどうか、さらにはひょっとしたら異なる文脈でつかわれていないかをチェックして警告を発します。

CONSTR.　ライブラリ・オブジェクトを適切に識別するために環境部の記述を正しく行うことは、とくに初心者にとってなかなか困難です。　一度テキストが ANALYZER に関して正しい状態になったら、生成された XML フォーマットは、どのオブジェクトの由来（origin）についても不明確なところのない情報を与えます。　時々エラー ***190**（アクセス不能な定理）が印字されます。

```
constr ArticleName:n
```

はアーティクル **ArticleName** から n 番目の定理を認識するために必要とされるすべてのコンストラクタのリストを与えます。　もし失われたファイル名だけのリストがほしければ

```
constr -f MyFile ArticleName:n
```

を実行するべきです。

実質上（virtually）すべての概念のコンストラクタのリストを得るため、**def, sch, exreg, funcreg** あるいは **condreg** キーワード（definition、 scheme, そして3種類のすべてのクラスタ登録）の前にナンバーをつけることができます。　最後の３つは自分でナンバーを付ける仕組み（numbering scheme）を持っていないので、あなたが自分で数えるか、あるいはもう一つの方法として（alternatively）ナンバーなしで上記コマンドを発行するしかないことに注意して下さい（この場合与えられた型の全てのオブジェクトを理解するのに必要なコンストラクタのリストが得られます）。

適切な環境指令があるように見えるのに、なぜ ANALYZER が ***103** エラーを返すのか、このコマンドを走らせることが理解の助けになることがよくあります。

4.6　エンハンサ（Enhancers）

どのMizarの配布も証明の質の改善について著者を助ける多くのプログラムを備えています;それらは単なる清書プログラムではありません。 このプログラムはどれも Mizar text について改善の可能性を提案します。

下のリストのユーティリティの呼び出し文字列（calling sequence）は大切です；それらは Emacs から「関係のない全てのユーティリティを実行する（'Execute all irrelevant utils'）」で実行できます、さもなければコマンドラインから

```
revf <enhancer> article
```

のように **revf** マクロを使います(エラー･フラグとその説明がテキストに追加されます)。

RELPREM. Mizarアーティクルを書く時に無関係の前提を捜索することは最も問題がなく人気のある調整（control）です。 RELPREM は文のジャスティフィケーションを受け入れるのに不必要な参照をチェックします；それはリンクと直接（straight forward）の両方の参照にマークをつけます。 それはまた（スキームに加えて）プレーン（訳注：スキーム構造を持たない、plain）の VERIFIER のエラーにも合致するので、標準的な VERIFIER に較べてチェックに要する時間は有意に長いのですが、アーティクルのベリフィケーションに使うのにも好都合です。

複数の RELPREM エラーが一つのジャスティフィケーションにある場合､それらのマークがついた参照のうちの少なくとも一つは削除が可能であることを意味します。 ***602** エラーは自分自身をオーバーライドすることもできます（can override themselves）。 ***602** と ***603** エラーは実質的に（virtually）同じ意味をもつことが起こり得ることを警戒して下さい － もし **then** と参照の両方が削除されるならば、以下に示す例のように推測（inference）はもうアクセプトされないでしょう。 マークされたエラー： 602, 603

```
A1: X <> {}; then
    X <> {} by A1;
::>          *603,602
```

RELINFER. 見直しをするソフトウェアのなかで唯一例外的に議論の多いのは無関係な推測、すなわち証明の中の不必要なステップを指摘するプログラムです（（訳注：ステップを除去した後では）参照は次のステップへ付けられます）。 それは証明を大幅に短縮することができますが、テキストの読みやすさ（readability）をかなり貧弱にする結果になります。 例えば証明にとって技術的に重要なある種の文は、無関係なステップであるというマークを付けられます、しかしそれらを除去するとユーザにとって潜在的に有用な補題への参照ではなく、同じライブラリへの参照（あるいはその両方(combination)）を強制的に繰り返させることになります；あるいはその除去はある意味で予想外（accidental）のことを起こします、すなわち人が証明を理解するのに決定的に重要なステップなのに機械には依然として不必要だということです（例えば、巻き戻し定義(unwinding definition) － 定義拡張(definitional expansions)）。 ここに証明の複雑さ（complexity）を減らす傾

向が人を誤らせることができるのを示します。

```
assume that
A: a = b and
B: b = c and
C: c = d;
D: a = c by A, B;
   a = d by C, D;
::>            *604
```

***604** エラーは参照リストの **D** をそのジャスティフィケーションすなわちラベル **A** と **B** で置き換える（replacing）ことを提案します。 すぐわかるように、そのやり方ではどのジャスティフィケーションもラベルの長いリストになりかねません、無意味な証明のステップなのに読む人にその証明の主要なステップであるという印象を与えるでしょう。
同様に ***605** はラベル付き文ではなく **then** を扱います。 エラー: 604, 605

RELITERS. 無関係なステップを見つけることは中間的な項をいくつか削除することで反復等号（iterative equalities）を短くするのに役立ちます。 特に、より長い項を最初に除去すべきだということを考慮に入れてステップを削除した場合には、通常は可読性に影響しません（システムはこれを無視します）。 エラー: 746

TRIVDEMO. Mizarの証明は階層構造（hierarchical）を持ちますが、時々上に述べた変換（transformation）の後に入れ子になった証明（nested proof）が無意味な（訳注：自明の）証明（trivial proofs）を検索するプログラムによって簡略化できることがあります（それは単純なジャスティフィケーション、すなわち、キーワード **by** に続く参照のリストに変えられ（reduced）ます）。
これは diffuse statement の場合にも起こります；そのときは、ユーザは自分で thesis を再生（reproduce）しなければなりません。 エラー: 607

CHKLAB. 通常、使用されていないラベルのチェックは証明の（一部の）完成と上で述べたプログラムの実行後に行われるべきです。 最もしばしば除去される未使用の前提はまさにライブラリ参照（すでに証明された定義と定理で MML にあります）です、しかし時々うっかりと（accidentally）局所的事例（a local fact）への参照が書かれます。 どの参照にも使われていないラベル付きの文があれば、CHKLAB はそのラベルは不必要だとマークしたと言ってよこします（pass）。 それでもなお（still though）証明のなかでその文への参照を次の行からの単純なリンクを通じて要求することができます（その場合は予約語 **then**）。 エラー: 601

INACC. アクセスできない（使用されていない）アーティクルの部分をラベルがつけら

れていないしリンクもされていない項目であると指摘することができます（証明のスケルトンの要素はエラーがあるというマークはされません）。

エラー: 610, 611（それぞれ使用されていないブロックの初めと終わりをマークします）

IRRVOC. **vocabulary** 指令からどの識別子を削除できるかをチェックします。

エラー: 709. これは ACCOMMODATOR エラーにつながることがあることに（can lead to）注意してください；このユーティリティは **notations** 指令にリストされているファイルを完全に削除してしまうかも知れません（それゆえエラー ***830**）、それはまたユーザにその不必要な宣言を削除するように強制します。

IRRTHS. **theorems** 指令と **schemes** 指令からどの識別子を除去できるかをチェックします。 エラー: 706（theorems）, 707（schemes）

5. Mizar数学ライブラリ（MIZAR MATHEMATICAL LIBRARY）

5.1 公理（AXIOMATICS）

5.1.1 *ファイル* HIDDEN（*file* HIDDEN）。 このアーティクルにMizarの公理の一部を文書化してあります ― それは集合論の根源部分（primitives）が Mizar Mathematical Library にどのように導入されているかを示します。 ここで定義された概念は標準的なベリフィケーションの対象ではないので、Mizarベリファイヤあるいは他のユーティリティでこのアーティクルを処理すればエラーが報告されます。

このアーティクルで原始**集合**（primitive **set**）：

```
definition
  mode set;
end;
```

を導入し、オブジェクトの同値性（equality）を述べる反射性（再帰性：reflexive）と対称性（symmetric）がサポートされる述語を導入します。

```
definition let x, y be set;
  pred x = y;
  reflexivity;
  symmetry;
end;
```

上に示したものの否定バージョンも一緒に導入します：

```
notation let x, y be set;
  antonym x <> y for x = y;
```

```
end;
```

最後に導入する原始型（primitives）は∈で、それは自動的に非対称性（asymmetry）の特性を獲得します。

```
definition let x, X be set;
  pred x in X;
  asymmetry;
end;
```

5.1.2 *ファイル* TARSKI（*file* TARSKI）。 このアーティクルはタルスキ・グロタンディック集合論（Tarski-Grothendieck set theory）の公理的基盤：外延性の公理（extensionality axiom）を定義します。

```
theorem :: TARSKI:2
  (for x holds x in X iff x in Y) implies X = Y;
```

ファンクタの定義の設定（settings）の中の対の公理（axiom of pair）（singleton、訳注：要素数１の集合{a}?）は後者から導かれるのですが、それもあります）

```
definition let y be set;
func { y } means
:: TARSKI:def 1
x in it iff x = y;
end;

definition let y, z be set;
  func { y, z } means
:: TARSKI:def 2
  x in it iff x = y or x = z;
commutativity;
end;

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

それから２つ別の公理が続きます：ユニオン（和集合）の公理（もう一度言います、定義

の構成型です）と正則性（regularity）の公理です。

```
definition let X be set;
  func union X means
:: TARSKI:def 4
  x in it iff ex Y st x in Y & Y in X;
end;
theorem :: TARSKI:7
  x in X implies ex Y st Y in X & not ex x st x in X & x in Y;
```

ZFC（訳注：Zermelo-Fraenkel 公理系に選択公理：axiom choice を加えたもの）のなかで置換公理のスキーム（scheme）の役割を演ずるフレンケルのスキーム（Fraenkel scheme）：

```
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;
```

クラトウスキ順序対（Kuratowski ordered pair）の定義は標準的な方法でジャスティファイすることができます。

```
definition let x, y be set;
  func [x,y] equals
:: TARSKI:def 5
  { { x,y }, { x } };
end;
```

そして、べき集合の公理、無限、選択、到達不能基数の存在（existence of inaccessible cardinals）を含意するタルスキの公理 A：

> 公理 A.　全ての集合 N について、次の条件群を満たす集合のシステム M が存在する：
> (i) $N \in M$
> (ii) $X \in M$ かつ $Y \subseteq X$ ならば $Y \in M$
> (iii) $X \in M$ かつ Z が X の部分集合の全てのシステムならば $Z \in M$
> (iv) $X \subseteq M$ かつ X と M の基数（濃度（potency））が等しくなければ $X \in M$：
> （Alfred Tarski, On well-ordered subsets of any set, Fundamenta Mathematicae, vol. 32 (1939), pp. 176-183.より採録）

```
theorem :: TARSKI:9
ex M st N in M &
 (for X,Y holds X in M & Y c= X implies Y in M) &
 (for X st X in M ex Z st Z in M & for Y st Y c= X holds Y in Z) &
 (for X holds X c= M implies X,M are_equipotent or X in M);
```

ここで等濃度（equipotency）はMizarの述語として：

```
definition let X, Y be set;
  pred X,Y are_equipotent means
:: TARSKI:def 6
    ex Z st
       (for x st x in X ex y st y in Y & [x,y] in Z) &
       (for y st y in Y ex x st x in X & [x,y] in Z) &
        for x,y,z,u st [x,y] in Z & [z,u] in Z holds x = z iff y = u;
 end;
```

と定義されています。

5.2 内容（Contents）

MML は大まかに５つに分けられます（配布物の中のファイル **mml.txt** に反映されています）：

— 公理系、前のセクションで述べたとおりです。
— 付録（addenda）、そのなかに技術的な性格を持ついくつかのアーティクル（論文）が収められています（例えばデデキントの切断による実数の構成）、通常はライブラリ委員会の著作です。
— EMM（Encyclopedia of Mathematics in Mizar：Mizarの数学百科事典）－注意深く選択された具体的なトピックについての定理、概念のコレクション（今までのところは主に集合のブール代数的特性、実数の特性、拡張された実数および複素数についての11ファイル）。
— requirements files －その中で **requirements** が標準的な方法で証明されていて、それらが提供を一時的に停止している自動操作についての短いファイル。
— regular articles.

MML の後半は実際には目に見えるようなブロックには分離していませんが、他より良く発達した小道を識別できます。 ここで集合論、一般位相空間論、束論（連続束論を含む）、関数解析、測度論を指摘することができます。

いままでのところ（As of now）、Wiedijk の“トップ 100 数学定理”から 53 の事例

（facts）が（訳注：Mizarで）形式化（formalized）されています[1]、それらは代数学の基本定理、オイラーの多面体の公式、ラムゼーの定理、シローの定理、そしてブラウエルの不動点定理です。　他の重要な定理のなかでは、MML の中にジョルダンの曲線定理、バーコフの多様体定理、ベルトランの前提条件、ジョンソンの束とモジュラー束についての定理、永田－スミルノフの定理、ケーニヒの補題、アレクサンダーの補題、ハーン－バナッハの定理、そしてウエダーバーンの定理です。
Association of Mizar Users のライブラリ委員会は改訂に責任があり、（査読者の示唆に基づいて）新しいアーティクルの採録を決定し、そして MML の新しい項目、例えば EMM ファイルや付録セクションなど、を決定します。

5.3 アーティクルの投稿（Submission of articles）

Mizarコミュニティが現在おもに取り組んでいるのは MML の開発と維持です。　ライブラリは 230人以上のアーティクルによる貢献をする著者の仕事の上に成立しています。原理的には誰でも MML を基礎にして自身のライブラリ構築を開始することができるのですが、現実にはほとんどの著者は相互関連した一連のアーティクルを開発するためだけのローカル・ライブラリを使います、そして形式化した定理が完成し次第、アーティクルを中央化してある MML へ投稿します。　この方法で彼らは、ライブラリをメンテナンスし一部を改訂して全般的な質と一貫性を向上させMizarソフトウェアやMizar言語の文法に変更があったときは常時アーティクルをアップデートしているライブラリ委員会の仕事を有効に利用することができます。　MML とのコンパチビリティーを維持しながらソフトウェアの変化に追いついていく一方で、分離されたデータベースをメンテナンスすることはかなり大変な仕事と思われます。
　アーティクルを MML に投稿する前に、著者は以下の基準にあっているかどうかチェックするべきです。

MML への収録をめざしているアーティクルはその中に十分な量の形式化された数学を持っていないといけません。　実際のところ短くすることはできません。　推薦されるアーティクルの長さは1200から1500行で、現行の MML のアーティクルの平均は2300行です。概して（as a rule）1000行以下のアーティクルはリジェクトされると思います。　アーティクルは一行80文字以下の制限を守るように要求されます。

---

[1] http://www.cs.ru.nl/~freek/100/ を見てください。

（1） MML アーティクルのファイル名は選択したリソースをインポートするため他のアーティクルの環境部で使われます。 それゆえ、システムは広いクラスのファイル名をサポートできるのですが、MML に投稿されるアーティクルは“**.miz**”拡張子持つ標準化“8+3”形式のファイル名の使用を強制されます。 ファイル名は文字、数字それからアンダースコア記号“_”だけを含まねばなりません。 最初の字は文字でないといけません（文字“x”は除外してください、これは特別な種類の*百科事典*のアーティクルのために予約されています）。 ファイル名の長さは5ないし8字ですが、8字が好まれます。 ファイル名は一意的（unique）でないといけません、すなわち既に MML で使用可能な全てのアーティクルの名前と異なっていなければなりません。 ファイル名はアーティクルのタイトル（と内容）に相当する略語であるべきです。

（2） MML に投稿されたアーティクルは（**prel** サブディレクトリにおいてある）ローカル・データベースに依存してはいけません、また MML にないアーティクルからビルトされてもいけません。 もしローカル・ライブラリが使用されるなら、（訳注：MML に入っていない）その準備段階のアーティクル（preliminaries）も一緒に投稿しないといけません。

（3） もしアーティクルがプライベート・ボキャブラリ・ファイル（Section 4.3 を見て下さい）を必要とするならそれも投稿原稿に入れるべきです。 ボキャブラリのファイル名は強制的に付加しなければならない“**.voc**”拡張子以外はアーティクルの名前と同じにします。 CHECKVOC ユーティリティを MML シンボルの重複を検出するために使うべきです（Section 4.5 を見て下さい）。 新しいシンボルは拡張 ASCII 文字を含んでいてはいけません。 1字のシンボルやシンボル名の中の空白はどちらも許されません。 著者はボキャブラリ・ファイルが使用しないシンボルを含んでいないことを確認してください。

（4） 投稿されたアーティクルは、いかなるベリフィケーションエラーも出さないようになっているのはもちろんのことです。 著者はアーティクルを書くときに使った複数の **@proof** を除去したことを最初に確認するべきです、それからMizarベリファイヤを呼びます。 現在使用可能な公式リリースMizarの最新バージョンを使うべきです。

アーティクルはMizar配布物で使用可能な標準的なユーティリティ：RELPREM, RELINFER, CHKLAB, INACC, TRIVDEMO そして RELITERS（Section 4.6 をみてください）に関して*クリーン*でなければなりません（訳注：エラーを出さないように改訂が済んでいること）。 IRRVOC と IRRTHS ツールを使ってアーティクルの環境部から不必要な指令を削除しなければなりません。

（5） 投稿されたアーティクルには書誌目録（bibliographic note）とサマリーを ***.bib** ファイルに入れたものを添付します（Mizar **doc** サブディレクトリにそういったフ

ァイルの例を見ることができます：**example.bib**）。　このファイルの名前は相当するMizarアーティクルの名前と同じものにします。　著者は example.bib ファイルにあるように形式化を実行するために使った外部ソース（すなわちMizarではないもの）を引用するように推奨されます。　ファイルの内容は英語でなければならないのに注意してください、それはMizarアーティクルを Formalized Mathematics Journal において自然言語表現に翻訳する過程で使われます、Section 5.4 を見て下さい。

（６） 投稿は最終的に著者が Mizar **doc** サブディレクトリにあるサブミッション・フォームを書き終えたときに完了します（著者が一人の場合は **mmldecl.txt**、もっと多数のときは **mmldecls.txt**）、そして標準の郵便か Fax で以下のアドレス：

Association of Mizar Users
University of Bialystok
Institute of Mathematics
ul. Akademicka 2
15-267 Bialystok
Poland
Fax: +48-85-745-75-45

に送付します。　これらの書式フォームの PDF バージョンがMizarウェブサイトでダウンロードできます。

（７） 貢献しようとする投稿ファイルは ZIP 圧縮アーカイブ・ファイルとして**mml@mizar.uwb.edu.pl** 宛ての e-mail の添付ファイルにしなければなりません。

5.4 形式化数学（Formalized Mathematics）

Mizarの著者が彼らのアーティクルを MML に投稿することによって得ることのできる利益は２重です。　以前のセクションで述べたようにアーティクルが MML に受理された場合ライブラリ委員会がそれのメンテナンスを始めますのでそのアーティクルは常に新しいバージョンのMizarソフトウェアと MML アーティクルの残りの部分とコンパチブルになります。　余分の利点としてはアーティクルのエクスポート可能な部分は翻訳ソフトウェアにより自動的に処理されます。　翻訳の結果は印刷物と電子フォーマットの両方で年４回発行されるジャーナル *Formalized Mathematics*[2] に掲載されます。　もちろん全ての MML に収録されようとするMizarアーティクルはMizarシステムでその正当性をチェックされますので、それらが論理的に健全であることは間違いありません。

---

[2] *Formalized Mathematics* のウェブサイト **fm.mizar.org** をみてください。

その内容の高い質を保証するため、全ての論文は関係のある分野の少なくとも３人のエキスパートにより査読されます。

英語への自動翻訳システムと LaTeX によるタイプセット（活字組）は（Andrzej Trybulec と Czesław Byliński による以前のデザインに基づき）Gregorz Bancerek がデザインと実装を行いました。 著者はそれが出版のために *Formalized Mathematics* 誌に受理されたら、どんな具合にそのアーティクルが自動的に LaTeX 化されるかを **fm.uwb.edu.pl/proof-read/** のオンライン校正システムでチェックできます。

6. Mizarについてのそれ以上の情報（MORE INFORMATION ON MIZAR）

本論文は基礎的なMizarの術語の実用的なリファレンス・マニュアルとして役立つよう意図されています。 読者はいまや MML の全てのMizarテキストを理解するための準備が終わって（//?? be be -> be）自分自身の証明をMizarで書いてベリファイする個別の実験を始めることができるはずです。 しかしMizarは複雑なシステムでその言語の記法は非常に豊富です、本論文で触れていない問題や疑問を生じるたくさんの面があることに疑いはありません。 ですから答えを Mizar Forum mailing list (**mizar.org/forum**) のなかで探すようアドバイスします、それはMizarの全ての面について全力で専念しています（devoted）。さらには献身的な e-mail ヘルプ・ライン、 Mizar User Service（**mus@mizar.uwb.edu.pl**）で、誰かがある特定の質問をするとエキスパートによる助言を見出すことができます。 有用な情報の偉大なソースが Mizar Wiki 共同計画（**wiki.mizar.org**）によって提供されていて、そこではMizarコミュニティが情報と専門的知識、アイデアを共有しています。

最後にここで本論文ではカバーしていないMizarに近い関係を持ついくつかのシステムについて言及しておきましょう。

－MML Query (a powerful semantic search engine for MML)
wiki.mizar.org
－MoMM(a matching, interreduction and database tool for mathematical databases optimized for Mizar)
wiki.mizar.org/twiki/bin/view/Mizar/MoMM
－Mizar Proof Advisor
wiki.mizar.org/twiki/bin/view/Mizar/MizarProofAdvisor

ライブラリ委員会は著者にこのプレビュー・システムを使うことを強く勧めています。

－MpTP (Mizar Problems for Theorem Proving)
wiki.mizar.org/twiki/bin/view/Mizar/MpTP
－Mizar mode for Emacs
wiki.mizar.org/twiki/bin/view/Mizar/MizarMode

APPENDIX

A. SKELETONS

A.1 Definitions

A.1.1 *Predicates*

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 assume Ψ(x_1, x_2, ..., x_n);
 pred π(x_1, x_2, ..., x_n) means :ident:
  Φ(x_1, x_2, ..., x_n);
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 assume Ψ(x_1, x_2, ..., x_n);
 pred π(x_1, x_2, ..., x_n) means :ident:
  Φ_1(x_1, x_2, ..., x_n) if Γ_1(x_1, x_2, ..., x_n),
  Φ_2(x_1, x_2, ..., x_n) if Γ_2(x_1, x_2, ..., x_n),
  Φ_3(x_1, x_2, ..., x_n) if Γ_3(x_1, x_2, ..., x_n)
  otherwise Φ_n(x_1, x_2, ..., x_n);
 consistency
 proof
  thus Γ_1(x_1, x_2, ..., x_n) & Γ_2(x_1, x_2, ..., x_n) implies
   (Φ_1(x_1, x_2, ..., x_n) iff Φ_2(x_1, x_2, ..., x_n));
  thus Γ_1(x_1, x_2, ..., x_n) & Γ_3(x_1, x_2, ..., x_n) implies
   (Φ_1(x_1, x_2, ..., x_n) iff Φ_3(x_1, x_2, ..., x_n));
  thus Γ_2(x_1, x_2, ..., x_n) & Γ_3(x_1, x_2, ..., x_n) implies
   (Φ_2(x_1, x_2, ..., x_n) iff Φ_3(x_1, x_2, ..., x_n));
 end;
end;
```

A.1.2 *Modes*

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 assume Ψ(x_1, x_2, ..., x_n);
 mode μ of x_1, x_2, ..., x_n -> Θ means :ident:
  Φ(x_1, x_2, ..., x_n, it);
 existence
 proof
  thus ex a being Θ st Φ(x_1, x_2, ..., x_n, a);
 end;
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 assume Ψ(x_1, x_2, ..., x_n);
 mode μ of x_1, x_2, ..., x_n -> Θ means :ident:
  Φ_1(x_1, x_2, ..., x_n, it) if Γ_1(x_1, x_2, ..., x_n),
```

```
  Φ2(x1, x2, ..., xn, it) if Γ2(x1, x2, ..., xn),
  Φ3(x1, x2, ..., xn, it) if Γ3(x1, x2, ..., xn)
  otherwise Φn(x1, x2, ..., xn, it);
 existence
 proof
  thus Γ1(x1, x2, ..., xn) implies ex a being Θ st Φ1(x1, x2, ..., xn, a);
  thus Γ2(x1, x2, ..., xn) implies ex a being Θ st Φ2(x1, x2, ..., xn, a);
  thus Γ3(x1, x2, ..., xn) implies ex a being Θ st Φ3(x1, x2, ..., xn, a);
  thus not Γ1(x1, x2, ..., xn) & not Γ2(x1, x2, ..., xn) & not Γ3(x1, x2, ..., xn)
   implies ex a being Θ st Φn(x1, x2, ..., xn, a);
 end;
 consistency
 proof
  let a be Θ;
  thus Γ1(x1, x2, ..., xn) & Γ2(x1, x2, ..., xn) implies
   (Φ1(x1, x2, ..., xn, a) iff Φ2(x1, x2, ..., xn, a));
  thus Γ1(x1, x2, ..., xn) & Γ3(x1, x2, ..., xn) implies
   (Φ1(x1, x2, ..., xn, a) iff Φ3(x1, x2, ..., xn, a));
  thus Γ2(x1, x2, ..., xn) & Γ3(x1, x2, ..., xn) implies
   (Φ2(x1, x2, ..., xn, a) iff Φ3(x1, x2, ..., xn, a));
 end;
end;
```

A.1.3 *Functors (means, equals)*

```
definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 assume Ψ(x1, x2, ..., xn);
 func ⊗(x1, x2, ..., xn) -> Θ means :ident:
  Φ(x1, x2, ..., xn, it);
 existence
 proof
  thus ex a being Θ st Φ(x1, x2, ..., xn, a);
 end;
 uniqueness
 proof
  thus for a,b being Θ st Φ(x1, x2, ..., xn, a) & Φ(x1, x2, ..., xn, b)
   holds a = b;
 end;
end;
```

```
definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 assume Ψ(x1, x2, ..., xn);
 func ⊗(x1, x2, ..., xn) -> Θ means :ident:
  Φ1(x1, x2, ..., xn, it) if Γ1(x1, x2, ..., xn),
  Φ2(x1, x2, ..., xn, it) if Γ2(x1, x2, ..., xn),
  Φ3(x1, x2, ..., xn, it) if Γ3(x1, x2, ..., xn)
  otherwise Φn(x1, x2, ..., xn, it);
```

```
 existence
 proof
  thus Γ1(x1,x2,...,xn) implies ex a being Θ st Φ1(x1,x2,...,xn,a);
  thus Γ2(x1,x2,...,xn) implies ex a being Θ st Φ2(x1,x2,...,xn,a);
  thus Γ3(x1,x2,...,xn) implies ex a being Θ st Φ3(x1,x2,...,xn,a);
  thus not Γ1(x1,x2,...,xn) & not Γ2(x1,x2,...,xn) & not Γ3(x1,x2,...,xn)
   implies ex a being Θ st Φn(x1,x2,...,xn,a);
 end;
 uniqueness
 proof
  let a,b be Θ;
  thus Γ1(x1,x2,...,xn) & Φ1(x1,x2,...,xn,a) & Φ1(x1,x2,...,xn,b)
   implies a = b;
  thus Γ2(x1,x2,...,xn) & Φ2(x1,x2,...,xn,a) & Φ2(x1,x2,...,xn,b)
   implies a = b;
  thus Γ3(x1,x2,...,xn) & Φ3(x1,x2,...,xn,a) & Φ3(x1,x2,...,xn,b)
   implies a = b;
  thus not Γ1(x1,x2,...,xn) & not Γ2(x1,x2,...,xn) & not Γ3(x1,x2,...,xn)
   & Φn(x1,x2,...,xn,a) & Φn(x1,x2,...,xn,b) implies a = b;
 end;
 consistency
 proof
  let a be Θ;
  thus Γ1(x1,x2,...,xn) & Γ2(x1,x2,...,xn) implies
   (Φ1(x1,x2,...,xn,a) iff Φ2(x1,x2,...,xn,a));
  thus Γ1(x1,x2,...,xn) & Γ3(x1,x2,...,xn) implies
   (Φ1(x1,x2,...,xn,a) iff Φ3(x1,x2,...,xn,a));
  thus Γ2(x1,x2,...,xn) & Γ3(x1,x2,...,xn) implies
   (Φ2(x1,x2,...,xn,a) iff Φ3(x1,x2,...,xn,a));
 end;
end;

definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 assume Ψ(x1,x2,...,xn);
 func ⊗(x1,x2,...,xn) -> Θ equals :ident:
  τ(x1,x2,...,xn);
 coherence
 proof
  thus τ(x1,x2,...,xn) is Θ;
 end;
end;

definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 assume Ψ(x1,x2,...,xn);
 func ⊗(x1,x2,...,xn) -> Θ equals :ident:
  τ1(x1,x2,...,xn) if Γ1(x1,x2,...,xn),
```

```
  τ2(x1, x2, ..., xn) if Γ2(x1, x2, ..., xn),
  τ3(x1, x2, ..., xn) if Γ3(x1, x2, ..., xn)
  otherwise τn(x1, x2, ..., xn);
 coherence
 proof
  thus Γ1(x1, x2, ..., xn) implies τ1(x1, x2, ..., xn) is Θ;
  thus Γ2(x1, x2, ..., xn) implies τ2(x1, x2, ..., xn) is Θ;
  thus Γ3(x1, x2, ..., xn) implies τ3(x1, x2, ..., xn) is Θ;
  thus not Γ1(x1, x2, ..., xn) & not Γ2(x1, x2, ..., xn) & not Γ3(x1, x2, ..., xn)
   implies τn(x1, x2, ..., xn) is Θ;
 end;
 consistency
 proof
  let a be Θ;
  thus Γ1(x1, x2, ..., xn) & Γ2(x1, x2, ..., xn) implies
   (a = τ1(x1, x2, ..., xn) iff a = τ2(x1, x2, ..., xn));
  thus Γ1(x1, x2, ..., xn) & Γ3(x1, x2, ..., xn) implies
   (a = τ1(x1, x2, ..., xn) iff a = τ3(x1, x2, ..., xn));
  thus Γ2(x1, x2, ..., xn) & Γ3(x1, x2, ..., xn) implies
   (a = τ2(x1, x2, ..., xn) iff a = τ3(x1, x2, ..., xn));
 end;
end;
```

A.1.4 *Attributes*

```
definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 assume Ψ(x1, x2, ..., xn);
 attr xn is α means :ident:
  Φ(x1, x2, ..., xn);
end;

definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 assume Ψ(x1, x2, ..., xn);
 attr xn is α means :ident:
  Φ1(x1, x2, ..., xn) if Γ1(x1, x2, ..., xn),
  Φ2(x1, x2, ..., xn) if Γ2(x1, x2, ..., xn),
  Φ3(x1, x2, ..., xn) if Γ3(x1, x2, ..., xn)
  otherwise Φn(x1, x2, ..., xn);
 consistency
 proof
  thus Γ1(x1, x2, ..., xn) & Γ2(x1, x2, ..., xn) implies
   (Φ1(x1, x2, ..., xn) iff Φ2(x1, x2, ..., xn));
  thus Γ1(x1, x2, ..., xn) & Γ3(x1, x2, ..., xn) implies
   (Φ1(x1, x2, ..., xn) iff Φ3(x1, x2, ..., xn));
  thus Γ2(x1, x2, ..., xn) & Γ3(x1, x2, ..., xn) implies
   (Φ2(x1, x2, ..., xn) iff Φ3(x1, x2, ..., xn));
 end;
```

```
end;
```

A.2 Redefinitions – result type is being changed

A.2.1 *Modes*

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 redefine mode μ of x_1,x_2,...,x_n -> Θ;
 coherence
 proof
  thus for a being μ of x_1,x_2,...,x_n holds a is Θ;
 end;
end;
```

A.2.2 *Functors*

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 redefine func ⊗(x_1,x_2,...,x_n) -> Θ;
 coherence
 proof
  thus ⊗(x_1,x_2,...,x_n) is Θ;
 end;
end;
```

A.3 Redefinitions – definiens is being changed

A.3.1 *Predicates*

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 assume Ψ(x_1,x_2,...,x_n);
 redefine pred π(x_1,x_2,...,x_n) means :ident:
  Φ(x_1,x_2,...,x_n);
 compatibility
 proof
  thus π(x_1,x_2,...,x_n) iff Φ(x_1,x_2,...,x_n);
 end;
end;
```

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 assume Ψ(x_1,x_2,...,x_n);
 redefine pred π(x_1,x_2,...,x_n) means :ident:
  Φ_1(x_1,x_2,...,x_n) if Γ_1(x_1,x_2,...,x_n),
  Φ_2(x_1,x_2,...,x_n) if Γ_2(x_1,x_2,...,x_n),
  Φ_3(x_1,x_2,...,x_n) if Γ_3(x_1,x_2,...,x_n)
  otherwise Φ_n(x_1,x_2,...,x_n);
 consistency;
 compatibility
 proof
```

```
  thus Γ1(x1, x2, ..., xn) implies (π(x1, x2, ..., xn) iff Φ1(x1, x2, ..., xn));
  thus Γ2(x1, x2, ..., xn) implies (π(x1, x2, ..., xn) iff Φ2(x1, x2, ..., xn));
  thus Γ3(x1, x2, ..., xn) implies (π(x1, x2, ..., xn) iff Φ3(x1, x2, ..., xn));
  thus not Γ1(x1, x2, ..., xn) & not Γ2(x1, x2, ..., xn) & not Γ3(x1, x2, ..., xn)
   implies (π(x1, x2, ..., xn) iff Φn(x1, x2, ..., xn));
 end;
end;
```

A.3.2 *Modes*

```
definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 assume Ψ(x1, x2, ..., xn);
 redefine mode μ of x1, x2, ..., xn means :ident:
  Φ(x1, x2, ..., xn, it);
 compatibility
 proof
  thus for a being set holds
   a is μ of x1, x2, ..., xn iff Φ(x1, x2, ..., xn, a);
 end;
end;

definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 assume Ψ(x1, x2, ..., xn);
 redefine mode μ of x1, x2, ..., xn means :ident:
  Φ1(x1, x2, ..., xn, it) if Γ1(x1, x2, ..., xn),
  Φ2(x1, x2, ..., xn, it) if Γ2(x1, x2, ..., xn),
  Φ3(x1, x2, ..., xn, it) if Γ3(x1, x2, ..., xn)
  otherwise Φn(x1, x2, ..., xn, it);
 compatibility
 proof
  let a be set;
  thus Γ1(x1, x2, ..., xn) implies
   (a is μ of x1, x2, ..., xn iff Φ1(x1, x2, ..., xn, a));
  thus Γ2(x1, x2, ..., xn) implies
   (a is μ of x1, x2, ..., xn iff Φ2(x1, x2, ..., xn, a));
  thus Γ3(x1, x2, ..., xn) implies
   (a is μ of x1, x2, ..., xn iff Φ3(x1, x2, ..., xn, a));
  thus not Γ1(x1, x2, ..., xn) & not Γ2(x1, x2, ..., xn) & not Γ3(x1, x2, ..., xn)
   implies (a is μ of x1, x2, ..., xn iff Φn(x1, x2, ..., xn, a));
 end;
 consistency;
end;
```

A.3.3 *Functors (means, equals)*

```
definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 assume Ψ(x1, x2, ..., xn);
```

```
 redefine func ⊗(x_1, x_2, ..., x_n) means :ident:
  Φ(x_1, x_2, ..., x_n, it);
 compatibility
 proof
  thus for a being Θ holds a = ⊗(x_1, x_2, ..., x_n) iff Φ(x_1, x_2, ..., x_n, a);
 end;
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 assume Ψ(x_1, x_2, ..., x_n);
 redefine func ⊗(x_1, x_2, ..., x_n) means :ident:
  Φ_1(x_1, x_2, ..., x_n, it) if Γ_1(x_1, x_2, ..., x_n),
  Φ_2(x_1, x_2, ..., x_n, it) if Γ_2(x_1, x_2, ..., x_n),
  Φ_3(x_1, x_2, ..., x_n, it) if Γ_3(x_1, x_2, ..., x_n)
  otherwise Φ_n(x_1, x_2, ..., x_n, it);
 consistency;
 compatibility
 proof
  let a be Θ;
  thus Γ_1(x_1, x_2, ..., x_n) implies
   (a = ⊗(x_1, x_2, ..., x_n) iff Φ_1(x_1, x_2, ..., x_n, a));
  thus Γ_2(x_1, x_2, ..., x_n) implies
   (a = ⊗(x_1, x_2, ..., x_n) iff Φ_2(x_1, x_2, ..., x_n, a));
  thus Γ_3(x_1, x_2, ..., x_n) implies
   (a = ⊗(x_1, x_2, ..., x_n) iff Φ_3(x_1, x_2, ..., x_n, a));
  thus not Γ_1(x_1, x_2, ..., x_n) & not Γ_2(x_1, x_2, ..., x_n) & not Γ_3(x_1, x_2, ..., x_n)
   implies (a = ⊗(x_1, x_2, ..., x_n) iff Φ_3(x_1, x_2, ..., x_n, a));
 end;
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 assume Ψ(x_1, x_2, ..., x_n);
 redefine func ⊗(x_1, x_2, ..., x_n) equals :ident:
  τ(x_1, x_2, ..., x_n);
 compatibility
 proof
  thus for a being Θ holds a = ⊗(x_1, x_2, ..., x_n) iff a = τ(x_1, x_2, ..., x_n);
 end;
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 assume Ψ(x_1, x_2, ..., x_n);
 redefine func ⊗(x_1, x_2, ..., x_n) equals :ident:
  τ_1(x_1, x_2, ..., x_n) if Γ_1(x_1, x_2, ..., x_n),
  τ_2(x_1, x_2, ..., x_n) if Γ_2(x_1, x_2, ..., x_n),
  τ_3(x_1, x_2, ..., x_n) if Γ_3(x_1, x_2, ..., x_n)
```

```
  otherwise τ_n(x_1, x_2, ..., x_n);
 consistency;
 compatibility
 proof
  let a be Θ;
  thus Γ_1(x_1, x_2, ..., x_n) implies
   (a = ⊗(x_1, x_2, ..., x_n) iff a = τ_1(x_1, x_2, ..., x_n));
  thus Γ_2(x_1, x_2, ..., x_n) implies
   (a = ⊗(x_1, x_2, ..., x_n) iff a = τ_2(x_1, x_2, ..., x_n));
  thus Γ_3(x_1, x_2, ..., x_n) implies
   (a = ⊗(x_1, x_2, ..., x_n) iff a = τ_3(x_1, x_2, ..., x_n));
  thus not Γ_1(x_1, x_2, ..., x_n) & not Γ_2(x_1, x_2, ..., x_n) & not Γ_3(x_1, x_2, ..., x_n)
   implies (a = ⊗(x_1, x_2, ..., x_n) iff a = τ_n(x_1, x_2, ..., x_n));
 end;
end;
```

A.3.4 *Attributes*

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 assume Ψ(x_1, x_2, ..., x_n);
 redefine attr x_n is α means :ident:
  Φ(x_1, x_2, ..., x_n);
 compatibility
 proof
  thus x_n is α iff Φ(x_1, x_2, ..., x_n);
 end;
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 assume Ψ(x_1, x_2, ..., x_n);
 redefine attr x_n is α means :ident:
  Φ_1(x_1, x_2, ..., x_n) if Γ_1(x_1, x_2, ..., x_n),
  Φ_2(x_1, x_2, ..., x_n) if Γ_2(x_1, x_2, ..., x_n),
  Φ_3(x_1, x_2, ..., x_n) if Γ_3(x_1, x_2, ..., x_n)
  otherwise Φ_n(x_1, x_2, ..., x_n);
 consistency;
 compatibility
 proof
  thus Γ_1(x_1, x_2, ..., x_n) implies (x_n is α iff Φ_1(x_1, x_2, ..., x_n));
  thus Γ_2(x_1, x_2, ..., x_n) implies (x_n is α iff Φ_2(x_1, x_2, ..., x_n));
  thus Γ_3(x_1, x_2, ..., x_n) implies (x_n is α iff Φ_3(x_1, x_2, ..., x_n));
  thus not Γ_1(x_1, x_2, ..., x_n) & not Γ_2(x_1, x_2, ..., x_n) &
   not Γ_3(x_1, x_2, ..., x_n) implies (x_n is α iff Φ_n(x_1, x_2, ..., x_n));
 end;
end;
```

A.4 Registrations

A.4.1 *Existential*

```
registration
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 cluster α_1 ... α_m Θ;
 existence
 proof
  thus ex a being Θ st a is α_1 ... α_m;
 end;
end;
```

A.4.2 *Conditional*

```
registration
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 cluster α_1 ... α_m -> α_{m+1} ... α_{m+1+k} Θ;
 coherence
 proof
  thus for a being Θ st a is α_1 ... α_m holds a is α_{m+1} ... α_{m+1+k};
 end;
end;
```

```
registration
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 cluster -> α_1 ... α_m Θ;
 coherence
 proof
  thus for a being Θ holds a is α_1 ... α_m;
 end;
end;
```

A.4.3 *Functorial*

```
registration
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 cluster τ(x_1, x_2, ..., x_n) -> α_1 ... α_m;
 coherence
 proof
  thus τ(x_1, x_2, ..., x_n) is α_1 ... α_m;
 end;
end;
```

```
registration
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 cluster τ(x_1, x_2, ..., x_n) -> α_1 ... α_m Θ;
 coherence
 proof
  thus for a being Θ st a = τ(x_1, x_2, ..., x_n) holds a is α_1 ... α_m;
 end;
end;
```

A.5 Properties in definitions

A.5.1 *Predicates.*

A.5.1.1 *Reflexivity*

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n, y_1,y_2 be Θ_{n+1};
 pred π(y_1,y_2) means :ident:
  Φ(x_1,x_2,...,x_n,y_1,y_2);
 reflexivity
 proof
  thus for a being Θ_{n+1} holds Φ(x_1,x_2,...,x_n,a,a);
 end;
end;
```

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n, y_1,y_2 be Θ_{n+1};
 pred π(y_1,y_2) means :ident:
  Φ_1(x_1,x_2,...,x_n,y_1,y_2) if Γ_1(x_1,x_2,...,x_n,y_1,y_2),
  Φ_2(x_1,x_2,...,x_n,y_1,y_2) if Γ_2(x_1,x_2,...,x_n,y_1,y_2),
  Φ_3(x_1,x_2,...,x_n,y_1,y_2) if Γ_3(x_1,x_2,...,x_n,y_1,y_2)
  otherwise Φ_n(x_1,x_2,...,x_n,y_1,y_2);
 consistency;
 reflexivity
 proof
  thus for a being Θ_{n+1} holds
   (Γ_1(x_1,x_2,...,x_n,a,a) implies Φ_1(x_1,x_2,...,x_n,a,a)) &
   (Γ_2(x_1,x_2,...,x_n,a,a) implies Φ_2(x_1,x_2,...,x_n,a,a)) &
   (Γ_3(x_1,x_2,...,x_n,a,a) implies Φ_3(x_1,x_2,...,x_n,a,a)) &
   (not Γ_1(x_1,x_2,...,x_n,a,a) & not Γ_2(x_1,x_2,...,x_n,a,a) &
    not Γ_3(x_1,x_2,...,x_n,a,a) implies Φ_n(x_1,x_2,...,x_n,a,a));
 end;
end;
```

A.5.1.2 *Irreflexivity*

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n, y_1,y_2 be Θ_{n+1};
 pred π(y_1,y_2) means :ident:
  Φ(x_1,x_2,...,x_n,y_1,y_2);
 irreflexivity
 proof
  thus for a being Θ_{n+1} holds not Φ(x_1,x_2,...,x_n,a,a);
 end;
end;
```

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n, y_1,y_2 be Θ_{n+1};
 pred π(y_1,y_2) means :ident:
  Φ_1(x_1,x_2,...,x_n,y_1,y_2) if Γ_1(x_1,x_2,...,x_n,y_1,y_2),
```

```
  Φ2(x1, x2, ..., xn, y1, y2) if Γ2(x1, x2, ..., xn, y1, y2),
  Φ3(x1, x2, ..., xn, y1, y2) if Γ3(x1, x2, ..., xn, y1, y2)
  otherwise Φn(x1, x2, ..., xn, y1, y2);
 consistency;
 irreflexivity
 proof
  thus for a being Θn+1 holds not
   (
    (Γ1(x1, x2, ..., xn, a, a) implies Φ1(x1, x2, ..., xn, a, a)) &
    (Γ2(x1, x2, ..., xn, a, a) implies Φ2(x1, x2, ..., xn, a, a)) &
    (Γ3(x1, x2, ..., xn, a, a) implies Φ3(x1, x2, ..., xn, a, a)) &
    (not Γ1(x1, x2, ..., xn, a, a) & not Γ2(x1, x2, ..., xn, a, a) &
     not Γ3(x1, x2, ..., xn, a, a) implies Φn(x1, x2, ..., xn, a, a))
   );
 end;
end;
```

A.5.1.3 *Symmetry*

```
definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn, y1, y2 be Θn+1;
 pred π(y1, y2) means :ident:
  Φ(x1, x2, ..., xn, y1, y2);
 symmetry
 proof
  thus for a, b being Θn+1 st Φ(x1, x2, ..., xn, a, b) holds
   Φ(x1, x2, ..., xn, b, a);
 end;
end;

definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn, y1, y2 be Θn+1;
 pred π(y1, y2) means :ident:
  Φ1(x1, x2, ..., xn, y1, y2) if Γ1(x1, x2, ..., xn, y1, y2),
  Φ2(x1, x2, ..., xn, y1, y2) if Γ2(x1, x2, ..., xn, y1, y2),
  Φ3(x1, x2, ..., xn, y1, y2) if Γ3(x1, x2, ..., xn, y1, y2)
  otherwise Φn(x1, x2, ..., xn, y1, y2);
 consistency;
 symmetry
 proof
  thus for a, b being Θn+1 st
   (
    (Γ1(x1, x2, ..., xn, a, b) implies Φ1(x1, x2, ..., xn, a, b)) &
    (Γ2(x1, x2, ..., xn, a, b) implies Φ2(x1, x2, ..., xn, a, b)) &
    (Γ3(x1, x2, ..., xn, a, b) implies Φ3(x1, x2, ..., xn, a, b)) &
    (not Γ1(x1, x2, ..., xn, a, b) & not Γ2(x1, x2, ..., xn, a, b) &
     not Γ3(x1, x2, ..., xn, a, b) implies Φn(x1, x2, ..., xn, a, b))
   ) holds
   (
```

```
    (Γ1(x1, x2, ..., xn, b, a) implies Φ1(x1, x2, ..., xn, b, a)) &
    (Γ2(x1, x2, ..., xn, b, a) implies Φ2(x1, x2, ..., xn, b, a)) &
    (Γ3(x1, x2, ..., xn, b, a) implies Φ3(x1, x2, ..., xn, b, a)) &
    (not Γ1(x1, x2, ..., xn, b, a) & not Γ2(x1, x2, ..., xn, b, a) &
     not Γ3(x1, x2, ..., xn, b, a) implies Φn(x1, x2, ..., xn, b, a))
   );
 end;
end;
```

A.5.1.4 *Asymmetry*

```
definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn, y1, y2 be Θn+1;
 pred π(y1, y2) means :ident:
  Φ(x1, x2, ..., xn, y1, y2);
 asymmetry
 proof
  thus for a, b being Θn+1 st Φ(x1, x2, ..., xn, a, b) holds
   not Φ(x1, x2, ..., xn, b, a);
 end;
end;

definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn, y1, y2 be Θn+1;
 pred π(y1, y2) means :ident:
  Φ1(x1, x2, ..., xn, y1, y2) if Γ1(x1, x2, ..., xn, y1, y2),
  Φ2(x1, x2, ..., xn, y1, y2) if Γ2(x1, x2, ..., xn, y1, y2),
  Φ3(x1, x2, ..., xn, y1, y2) if Γ3(x1, x2, ..., xn, y1, y2)
  otherwise Φn(x1, x2, ..., xn, y1, y2);
 consistency;
 asymmetry
 proof
  thus for a, b being Θn+1 st
   (
    (Γ1(x1, x2, ..., xn, a, b) implies Φ1(x1, x2, ..., xn, a, b)) &
    (Γ2(x1, x2, ..., xn, a, b) implies Φ2(x1, x2, ..., xn, a, b)) &
    (Γ3(x1, x2, ..., xn, a, b) implies Φ3(x1, x2, ..., xn, a, b)) &
    (not Γ1(x1, x2, ..., xn, a, b) & not Γ2(x1, x2, ..., xn, a, b) &
     not Γ3(x1, x2, ..., xn, a, b) implies Φn(x1, x2, ..., xn, a, b))
   ) holds not
   (
    (Γ1(x1, x2, ..., xn, b, a) implies Φ1(x1, x2, ..., xn, b, a)) &
    (Γ2(x1, x2, ..., xn, b, a) implies Φ2(x1, x2, ..., xn, b, a)) &
    (Γ3(x1, x2, ..., xn, b, a) implies Φ3(x1, x2, ..., xn, b, a)) &
    (not Γ1(x1, x2, ..., xn, b, a) & not Γ2(x1, x2, ..., xn, b, a) &
     not Γ3(x1, x2, ..., xn, b, a) implies Φn(x1, x2, ..., xn, b, a))
   );
 end;
end;
```

A.5.1.5 *Connectedness*

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n, y_1,y_2 be Θ_{n+1};
 pred π(y_1,y_2) means :ident:
  Φ(x_1,x_2,...,x_n,y_1,y_2);
 connectedness
 proof
  thus for a, b being Θ_{n+1} holds Φ(x_1,x_2,...,x_n,a,b) or
   Φ(x_1,x_2,...,x_n,b,a);
 end;
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n, y_1,y_2 be Θ_{n+1};
 pred π(y_1,y_2) means :ident:
  Φ_1(x_1,x_2,...,x_n,y_1,y_2) if Γ_1(x_1,x_2,...,x_n,y_1,y_2),
  Φ_2(x_1,x_2,...,x_n,y_1,y_2) if Γ_2(x_1,x_2,...,x_n,y_1,y_2),
  Φ_3(x_1,x_2,...,x_n,y_1,y_2) if Γ_3(x_1,x_2,...,x_n,y_1,y_2)
  otherwise Φ_n(x_1,x_2,...,x_n,y_1,y_2);
 consistency;
 connectedness
 proof
  thus for a, b being Θ_{n+1} holds
   (
    (Γ_1(x_1,x_2,...,x_n,a,b) implies Φ_1(x_1,x_2,...,x_n,a,b)) &
    (Γ_2(x_1,x_2,...,x_n,a,b) implies Φ_2(x_1,x_2,...,x_n,a,b)) &
    (Γ_3(x_1,x_2,...,x_n,a,b) implies Φ_3(x_1,x_2,...,x_n,a,b)) &
    (not Γ_1(x_1,x_2,...,x_n,a,b) & not Γ_2(x_1,x_2,...,x_n,a,b) &
     not Γ_3(x_1,x_2,...,x_n,a,b) implies Φ_n(x_1,x_2,...,x_n,a,b))
   ) or
   (
    (Γ_1(x_1,x_2,...,x_n,b,a) implies Φ_1(x_1,x_2,...,x_n,b,a)) &
    (Γ_2(x_1,x_2,...,x_n,b,a) implies Φ_2(x_1,x_2,...,x_n,b,a)) &
    (Γ_3(x_1,x_2,...,x_n,b,a) implies Φ_3(x_1,x_2,...,x_n,b,a)) &
    (not Γ_1(x_1,x_2,...,x_n,b,a) & not Γ_2(x_1,x_2,...,x_n,b,a) &
     not Γ_3(x_1,x_2,...,x_n,b,a) implies Φ_n(x_1,x_2,...,x_n,b,a))
   );
 end;
end;
```

A.5.2 *Functors.*

A.5.2.1 *Involutiveness (means, equals)*

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 func ⊗(x_n) -> Θ_n means :ident:
  Φ(x_1,x_2,...,x_n,it);
 existence;
```

```
 uniqueness;
 involutiveness
 proof
  thus for a, b being Θ_n st Φ(x_1, x_2, ..., x_{n-1}, b, a)
   holds Φ(x_1, x_2, ..., x_{n-1}, a, b);
 end;
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 func ⊗(x_n) -> Θ_n means :ident:
  Φ_1(x_1, x_2, ..., x_n, it) if Γ_1(x_1, x_2, ..., x_n),
  Φ_2(x_1, x_2, ..., x_n, it) if Γ_2(x_1, x_2, ..., x_n),
  Φ_3(x_1, x_2, ..., x_n, it) if Γ_3(x_1, x_2, ..., x_n)
  otherwise Φ_n(x_1, x_2, ..., x_n, it);
 existence;
 uniqueness;
 consistency;
 involutiveness
 proof
  thus for a, b being Θ_n st
   (
    (Γ_1(x_1, x_2, ..., x_{n-1}, b) implies Φ_1(x_1, x_2, ..., x_{n-1}, b, a)) &
    (Γ_2(x_1, x_2, ..., x_{n-1}, b) implies Φ_2(x_1, x_2, ..., x_{n-1}, b, a)) &
    (Γ_3(x_1, x_2, ..., x_{n-1}, b) implies Φ_3(x_1, x_2, ..., x_{n-1}, b, a)) &
    (not Γ_1(x_1, x_2, ..., x_{n-1}, b) & not Γ_2(x_1, x_2, ..., x_{n-1}, b) &
     not Γ_3(x_1, x_2, ..., x_{n-1}, b) implies Φ_n(x_1, x_2, ..., x_{n-1}, b, a))
   ) holds
   (
    (Γ_1(x_1, x_2, ..., x_{n-1}, a) implies Φ_1(x_1, x_2, ..., x_{n-1}, a, b)) &
    (Γ_2(x_1, x_2, ..., x_{n-1}, a) implies Φ_2(x_1, x_2, ..., x_{n-1}, a, b)) &
    (Γ_3(x_1, x_2, ..., x_{n-1}, a) implies Φ_3(x_1, x_2, ..., x_{n-1}, a, b)) &
    (not Γ_1(x_1, x_2, ..., x_{n-1}, a) & not Γ_2(x_1, x_2, ..., x_{n-1}, a) &
     not Γ_3(x_1, x_2, ..., x_{n-1}, a) implies Φ_n(x_1, x_2, ..., x_{n-1}, a, b))
   );
 end;
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 func ⊗(x_n) -> Θ_n equals :ident:
  τ(x_1, x_2, ..., x_n);
 coherence;
 involutiveness
 proof
  thus for a, b being Θ_n st a = τ(x_1, x_2, ..., x_{n-1}, b) holds
   b = τ(x_1, x_2, ..., x_{n-1}, a);
 end;
```

```
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 func ⊗(x_n) -> Θ_n equals :ident:
  τ_1(x_1, x_2, ..., x_n) if Γ_1(x_1, x_2, ..., x_n),
  τ_2(x_1, x_2, ..., x_n) if Γ_2(x_1, x_2, ..., x_n),
  τ_3(x_1, x_2, ..., x_n) if Γ_3(x_1, x_2, ..., x_n)
  otherwise τ_n(x_1, x_2, ..., x_n);
 coherence;
 consistency;
 involutiveness
 proof
  for a, b being Θ_n st
   (Γ_1(x_1, x_2, ..., x_{n-1}, b) implies a = τ_1(x_1, x_2, ..., x_{n-1}, b)) &
   (Γ_2(x_1, x_2, ..., x_{n-1}, b) implies a = τ_2(x_1, x_2, ..., x_{n-1}, b)) &
   (Γ_3(x_1, x_2, ..., x_{n-1}, b) implies a = τ_3(x_1, x_2, ..., x_{n-1}, b)) &
   (not Γ_1(x_1, x_2, ..., x_{n-1}, b) & not Γ_2(x_1, x_2, ..., x_{n-1}, b) &
     not Γ_3(x_1, x_2, ..., x_{n-1}, b) implies a = τ_n(x_1, x_2, ..., x_{n-1}, b))
  holds
   (Γ_1(x_1, x_2, ..., x_{n-1}, a) implies b = τ_1(x_1, x_2, ..., x_{n-1}, a)) &
   (Γ_2(x_1, x_2, ..., x_{n-1}, a) implies b = τ_2(x_1, x_2, ..., x_{n-1}, a)) &
   (Γ_3(x_1, x_2, ..., x_{n-1}, a) implies b = τ_3(x_1, x_2, ..., x_{n-1}, a)) &
   (not Γ_1(x_1, x_2, ..., x_{n-1}, a) & not Γ_2(x_1, x_2, ..., x_{n-1}, a) &
     not Γ_3(x_1, x_2, ..., x_{n-1}, a) implies b = τ_n(x_1, x_2, ..., x_{n-1}, a));
 end;
end;
```

A.5.2.2 *Projectivity (means, equals)*

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 func ⊗(x_n) -> Θ_{n+1} means :ident:
  Φ(x_1, x_2, ..., x_n, it);
 existence;
 uniqueness;
 projectivity
 proof
  thus for a, b being Θ_{n+1} st Φ(x_1, x_2, ..., x_{n-1}, b, a) holds
   Φ(x_1, x_2, ..., x_{n-1}, a, a);
 end;
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 func ⊗(x_n) -> Θ_{n+1} means :ident:
  Φ_1(x_1, x_2, ..., x_n, it) if Γ_1(x_1, x_2, ..., x_n),
  Φ_2(x_1, x_2, ..., x_n, it) if Γ_2(x_1, x_2, ..., x_n),
  Φ_3(x_1, x_2, ..., x_n, it) if Γ_3(x_1, x_2, ..., x_n)
```

```
  otherwise Φ_n(x_1, x_2, ..., x_n, it);
 existence;
 uniqueness;
 consistency;
 projectivity
 proof
  thus for a, b being Θ_{n+1} st
   (
    (Γ_1(x_1, x_2, ..., x_{n-1}, b) implies Φ_1(x_1, x_2, ..., x_{n-1}, b, a)) &
    (Γ_2(x_1, x_2, ..., x_{n-1}, b) implies Φ_2(x_1, x_2, ..., x_{n-1}, b, a)) &
    (Γ_3(x_1, x_2, ..., x_{n-1}, b) implies Φ_3(x_1, x_2, ..., x_{n-1}, b, a)) &
    (not Γ_1(x_1, x_2, ..., x_{n-1}, b) & not Γ_2(x_1, x_2, ..., x_{n-1}, b) &
     not Γ_3(x_1, x_2, ..., x_{n-1}, b) implies Φ_n(x_1, x_2, ..., x_{n-1}, b, a))
   ) holds
   (
    (Γ_1(x_1, x_2, ..., x_{n-1}, a) implies Φ_1(x_1, x_2, ..., x_{n-1}, a, a)) &
    (Γ_2(x_1, x_2, ..., x_{n-1}, a) implies Φ_2(x_1, x_2, ..., x_{n-1}, a, a)) &
    (Γ_3(x_1, x_2, ..., x_{n-1}, a) implies Φ_3(x_1, x_2, ..., x_{n-1}, a, a)) &
    (not Γ_1(x_1, x_2, ..., x_{n-1}, a) & not Γ_2(x_1, x_2, ..., x_{n-1}, a) &
     not Γ_3(x_1, x_2, ..., x_{n-1}, a) implies Φ_n(x_1, x_2, ..., x_{n-1}, a, a))
   );
 end;
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 func ⊗(x_n) -> Θ_{n+1} equals :ident:
  τ(x_1, x_2, ..., x_n);
 coherence;
 projectivity
 proof
  thus for a, b being Θ_{n+1} st a = τ(x_1, x_2, ..., x_{n-1}, b) holds
   a = τ(x_1, x_2, ..., x_{n-1}, a);
 end;
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 func ⊗(x_n) -> Θ_{n+1} equals :ident:
  τ_1(x_1, x_2, ..., x_n) if Γ_1(x_1, x_2, ..., x_n),
  τ_2(x_1, x_2, ..., x_n) if Γ_2(x_1, x_2, ..., x_n),
  τ_3(x_1, x_2, ..., x_n) if Γ_3(x_1, x_2, ..., x_n)
  otherwise τ_n(x_1, x_2, ..., x_n);
 coherence;
 consistency;
 projectivity
 proof
  thus for a, b being Θ_{n+1} st
```

```
   (
    (Γ1(x1, x2, ..., xn−1, b) implies a = τ1(x1, x2, ..., xn−1, b)) &
    (Γ2(x1, x2, ..., xn−1, b) implies a = τ2(x1, x2, ..., xn−1, b)) &
    (Γ3(x1, x2, ..., xn−1, b) implies a = τ3(x1, x2, ..., xn−1, b)) &
    (not Γ1(x1, x2, ..., xn−1, b) & not Γ2(x1, x2, ..., xn−1, b) &
     not Γ3(x1, x2, ..., xn−1, b) implies a = τn(x1, x2, ..., xn−1, b))
   ) holds
   (
    (Γ1(x1, x2, ..., xn−1, a) implies a = τ1(x1, x2, ..., xn−1, a)) &
    (Γ2(x1, x2, ..., xn−1, a) implies a = τ2(x1, x2, ..., xn−1, a)) &
    (Γ3(x1, x2, ..., xn−1, a) implies a = τ3(x1, x2, ..., xn−1, a)) &
    (not Γ1(x1, x2, ..., xn−1, a) & not Γ2(x1, x2, ..., xn−1, a) &
     not Γ3(x1, x2, ..., xn−1, a) implies a = τn(x1, x2, ..., xn−1, a))
   );
 end;
end;
```

A.5.2.3 *Idempotence (means, equals)*

```
definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn, y1, y2 be Θn+1;
 func ⊗(y1, y2) -> Θn+2 means :ident:
  Φ(x1, x2, ..., xn, y1, y2, it);
 existence;
 uniqueness;
 idempotence
 proof
  thus for a being Θn+1 holds Φ(x1, x2, ..., xn, a, a, a);
 end;
end;

definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn, y1, y2 be Θn+1;
 func ⊗(y1, y2) -> Θn+2 means :ident:
  Φ1(x1, x2, ..., xn, y1, y2, it) if Γ1(x1, x2, ..., xn, y1, y2),
  Φ2(x1, x2, ..., xn, y1, y2, it) if Γ2(x1, x2, ..., xn, y1, y2),
  Φ3(x1, x2, ..., xn, y1, y2, it) if Γ3(x1, x2, ..., xn, y1, y2)
  otherwise Φn(x1, x2, ..., xn, y1, y2, it);
 existence;
 uniqueness;
 consistency;
 idempotence
 proof
  thus for a being Θn+1 holds
   (Γ1(x1, x2, ..., xn, a, a) implies Φ1(x1, x2, ..., xn, a, a, a)) &
   (Γ2(x1, x2, ..., xn, a, a) implies Φ2(x1, x2, ..., xn, a, a, a)) &
   (Γ3(x1, x2, ..., xn, a, a) implies Φ3(x1, x2, ..., xn, a, a, a)) &
   (not Γ1(x1, x2, ..., xn, a, a) & not Γ2(x1, x2, ..., xn, a, a) &
    not Γ3(x1, x2, ..., xn, a, a) implies Φn(x1, x2, ..., xn, a, a, a));
```

```
 end;
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n, y_1, y_2 be Θ_{n+1};
 func ⊗(y_1, y_2) -> Θ_{n+2} equals :ident:
  τ(x_1, x_2, ..., x_n, y_1, y_2);
 coherence;
 idempotence
 proof
  thus for a being Θ_{n+1} holds a = τ(x_1, x_2, ..., x_n, a, a);
 end;
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n, y_1, y_2 be Θ_{n+1};
 func ⊗(y_1, y_2) -> Θ_{n+2} equals :ident:
  τ_1(x_1, x_2, ..., x_n, y_1, y_2) if Γ_1(x_1, x_2, ..., x_n, y_1, y_2),
  τ_2(x_1, x_2, ..., x_n, y_1, y_2) if Γ_2(x_1, x_2, ..., x_n, y_1, y_2),
  τ_3(x_1, x_2, ..., x_n, y_1, y_2) if Γ_3(x_1, x_2, ..., x_n, y_1, y_2)
  otherwise τ_n(x_1, x_2, ..., x_n, y_1, y_2);
 coherence;
 consistency;
 idempotence
 proof
  thus for a being Θ_{n+1} holds
   (Γ_1(x_1, x_2, ..., x_n, a, a) implies a = τ_1(x_1, x_2, ..., x_n, a, a)) &
   (Γ_2(x_1, x_2, ..., x_n, a, a) implies a = τ_2(x_1, x_2, ..., x_n, a, a)) &
   (Γ_3(x_1, x_2, ..., x_n, a, a) implies a = τ_3(x_1, x_2, ..., x_n, a, a)) &
   (not Γ_1(x_1, x_2, ..., x_n, a, a) & not Γ_2(x_1, x_2, ..., x_n, a, a) &
    not Γ_3(x_1, x_2, ..., x_n, a, a) implies a = τ_n(x_1, x_2, ..., x_n, a, a));
 end;
end;
```

A.5.2.4 *Commutativity (means, equals)*

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n, y_1, y_2 be Θ_{n+1};
 func ⊗(y_1, y_2) -> Θ_{n+2} means :ident:
  Φ(x_1, x_2, ..., x_n, y_1, y_2, it);
 existence;
 uniqueness;
 commutativity
 proof
  thus for a being Θ_{n+2}, b, c being Θ_{n+1} st Φ(x_1, x_2, ..., x_n, b, c, a)
   holds Φ(x_1, x_2, ..., x_n, c, b, a);
 end;
end;

definition
```

```
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n, y_1,y_2 be Θ_{n+1};
 func ⊗(y_1,y_2) -> Θ_{n+2} means :ident:
  Φ_1(x_1,x_2,...,x_n,y_1,y_2,it) if Γ_1(x_1,x_2,...,x_n,y_1,y_2),
  Φ_2(x_1,x_2,...,x_n,y_1,y_2,it) if Γ_2(x_1,x_2,...,x_n,y_1,y_2),
  Φ_3(x_1,x_2,...,x_n,y_1,y_2,it) if Γ_3(x_1,x_2,...,x_n,y_1,y_2)
  otherwise Φ_n(x_1,x_2,...,x_n,y_1,y_2,it);
 existence;
 uniqueness;
 commutativity
 proof
  thus for a being Θ_{n+2}, b, c being Θ_{n+1} st
   (
    (Γ_1(x_1,x_2,...,x_n,b,c) implies Φ_1(x_1,x_2,...,x_n,b,c,a)) &
    (Γ_2(x_1,x_2,...,x_n,b,c) implies Φ_2(x_1,x_2,...,x_n,b,c,a)) &
    (Γ_3(x_1,x_2,...,x_n,b,c) implies Φ_3(x_1,x_2,...,x_n,b,c,a)) &
    (not Γ_1(x_1,x_2,...,x_n,b,c) & not Γ_2(x_1,x_2,...,x_n,b,c) &
     not Γ_3(x_1,x_2,...,x_n,b,c) implies Φ_n(x_1,x_2,...,x_n,b,c,a))
   ) holds
   (
    (Γ_1(x_1,x_2,...,x_n,c,b) implies Φ_1(x_1,x_2,...,x_n,c,b,a)) &
    (Γ_2(x_1,x_2,...,x_n,c,b) implies Φ_2(x_1,x_2,...,x_n,c,b,a)) &
    (Γ_3(x_1,x_2,...,x_n,c,b) implies Φ_3(x_1,x_2,...,x_n,c,b,a)) &
    (not Γ_1(x_1,x_2,...,x_n,c,b) & not Γ_2(x_1,x_2,...,x_n,c,b) &
     not Γ_3(x_1,x_2,...,x_n,c,b) implies Φ_n(x_1,x_2,...,x_n,c,b,a))
   );
 end;
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n, y_1,y_2 be Θ_{n+1};
 func ⊗(y_1,y_2) -> Θ_{n+2} equals :ident:
  τ(x_1,x_2,...,x_n,y_1,y_2);
 coherence;
 commutativity
 proof
  thus for a being Θ_{n+2}, b, c being Θ_{n+1} st a = τ(x_1,x_2,...,x_n,b,c)
   holds a = τ(x_1,x_2,...,x_n,c,b);
 end;
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n, y_1,y_2 be Θ_{n+1};
 func ⊗(y_1,y_2) -> Θ_{n+2} equals :ident:
  τ_1(x_1,x_2,...,x_n,y_1,y_2) if Γ_1(x_1,x_2,...,x_n,y_1,y_2),
  τ_2(x_1,x_2,...,x_n,y_1,y_2) if Γ_2(x_1,x_2,...,x_n,y_1,y_2),
  τ_3(x_1,x_2,...,x_n,y_1,y_2) if Γ_3(x_1,x_2,...,x_n,y_1,y_2)
  otherwise τ_n(x_1,x_2,...,x_n,y_1,y_2);
 coherence;
```

```
 consistency;
 commutativity
 proof
  thus for a being Θ_{n+2}, b, c being Θ_{n+1} st
   (
    (Γ_1(x_1, x_2, ..., x_n, b, c) implies a = τ_1(x_1, x_2, ..., x_n, b, c)) &
    (Γ_2(x_1, x_2, ..., x_n, b, c) implies a = τ_2(x_1, x_2, ..., x_n, b, c)) &
    (Γ_3(x_1, x_2, ..., x_n, b, c) implies a = τ_3(x_1, x_2, ..., x_n, b, c)) &
    (not Γ_1(x_1, x_2, ..., x_n, b, c) & not Γ_2(x_1, x_2, ..., x_n, b, c) &
     not Γ_3(x_1, x_2, ..., x_n, b, c) implies a = τ_n(x_1, x_2, ..., x_n, b, c))
   ) holds
   (
    (Γ_1(x_1, x_2, ..., x_n, c, b) implies a = τ_1(x_1, x_2, ..., x_n, c, b)) &
    (Γ_2(x_1, x_2, ..., x_n, c, b) implies a = τ_2(x_1, x_2, ..., x_n, c, b)) &
    (Γ_3(x_1, x_2, ..., x_n, c, b) implies a = τ_3(x_1, x_2, ..., x_n, c, b)) &
    (not Γ_1(x_1, x_2, ..., x_n, c, b) & not Γ_2(x_1, x_2, ..., x_n, c, b) &
     not Γ_3(x_1, x_2, ..., x_n, c, b) implies a = τ_n(x_1, x_2, ..., x_n, c, b))
   );
 end;
end;
```

A.6 Properties in redefinitions

A.6.1 *Predicates.*

A.6.1.1 *Reflexivity*

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n, y_1, y_2 be Θ_{n+1};
 redefine pred π(y_1, y_2);
 reflexivity
 proof
  thus for a being Θ_{n+1} holds π(a, a);
 end;
end;
```

A.6.1.2 *Irreflexivity*

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n, y_1, y_2 be Θ_{n+1};
 redefine pred π(y_1, y_2);
 irreflexivity
 proof
  thus for a being Θ_{n+1} holds not π(a, a);
 end;
end;
```

A.6.1.3 *Symmetry*

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n, y_1, y_2 be Θ_{n+1};
```

```
 redefine pred π(y1, y2);
 symmetry
 proof
  thus for a, b being Θn+1 st π(a, b) holds π(b, a);
 end;
end;
```

A.6.1.4 *Asymmetry*

```
definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn, y1, y2 be Θn+1;
 redefine pred π(y1, y2);
 asymmetry
 proof
  thus for a, b being Θn+1 st π(a, b) holds not π(b, a);
 end;
end;
```

A.6.1.5 *Connectedness*

```
definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn, y1, y2 be Θn+1;
 redefine pred π(y1, y2);
 connectedness
 proof
  thus for a, b being Θn+1 holds π(a, b) or π(b, a);
 end;
end;
```

A.6.2 *Functors.*

A.6.2.1 *Commutativity*

```
definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn, y1, y2 be Θn+1;
 redefine func ⊗(y1, y2);
 commutativity
 proof
  thus for a, b being Θn+1 holds ⊗(a, b) = ⊗(b, a);
 end;
end;
```

A.7 Definitional expansions

A.7.1 *Predicates.*

A.7.1.1 *Non permissive*

```
definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 pred π(x1, x2, ..., xn) means :ident:
  Φ(x1, x2, ..., xn);
end;
```

$\pi(x_1, x_2, \ldots, x_n)$
proof
 thus $\Phi(x_1, x_2, \ldots, x_n)$;
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 pred $\pi(x_1, x_2, \ldots, x_n)$ means :*ident*:
  $\Phi_1(x_1, x_2, \ldots, x_n)$ if $\Gamma_1(x_1, x_2, \ldots, x_n)$,
  $\Phi_2(x_1, x_2, \ldots, x_n)$ if $\Gamma_2(x_1, x_2, \ldots, x_n)$,
  $\Phi_3(x_1, x_2, \ldots, x_n)$ if $\Gamma_3(x_1, x_2, \ldots, x_n)$
  otherwise $\Phi_n(x_1, x_2, \ldots, x_n)$;
 consistency;
end;

$\pi(x_1, x_2, \ldots, x_n)$
proof
 per cases;
 case $\Gamma_1(x_1, x_2, \ldots, x_n)$;
  thus $\Phi_1(x_1, x_2, \ldots, x_n)$;
 end;
 case $\Gamma_2(x_1, x_2, \ldots, x_n)$;
  thus $\Phi_2(x_1, x_2, \ldots, x_n)$;
 end;
 case $\Gamma_3(x_1, x_2, \ldots, x_n)$;
  thus $\Phi_3(x_1, x_2, \ldots, x_n)$;
 end;
 case not $\Gamma_1(x_1, x_2, \ldots, x_n)$ & not $\Gamma_2(x_1, x_2, \ldots, x_n)$ & not $\Gamma_3(x_1, x_2, \ldots, x_n)$;
  thus $\Phi_n(x_1, x_2, \ldots, x_n)$;
 end;
end;

A.7.1.2 *Permissive*

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 assume $\Psi(x_1, x_2, \ldots, x_n)$;
 pred $\pi(x_1, x_2, \ldots, x_n)$ means :*ident*:
  $\Phi(x_1, x_2, \ldots, x_n)$;
end;

$\pi(x_1, x_2, \ldots, x_n)$
proof
 thus $\Psi(x_1, x_2, \ldots, x_n)$;
 thus $\Phi(x_1, x_2, \ldots, x_n)$;
end;

definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 assume $\Psi(x_1, x_2, \ldots, x_n)$;

```
 pred π(x1, x2, ..., xn) means :ident:
  Φ1(x1, x2, ..., xn) if Γ1(x1, x2, ..., xn),
  Φ2(x1, x2, ..., xn) if Γ2(x1, x2, ..., xn),
  Φ3(x1, x2, ..., xn) if Γ3(x1, x2, ..., xn)
  otherwise Φn(x1, x2, ..., xn);
 consistency;
end;

π(x1, x2, ..., xn)
proof
 thus Ψ(x1, x2, ..., xn);
 per cases;
 case Γ1(x1, x2, ..., xn);
  thus Φ1(x1, x2, ..., xn);
 end;
 case Γ2(x1, x2, ..., xn);
  thus Φ2(x1, x2, ..., xn);
 end;
 case Γ3(x1, x2, ..., xn);
  thus Φ3(x1, x2, ..., xn);
 end;
 case not Γ1(x1, x2, ..., xn) & not Γ2(x1, x2, ..., xn) & not Γ3(x1, x2, ..., xn);
  thus Φn(x1, x2, ..., xn);
 end;
end;
```

A.7.2 *Modes.*

A.7.2.1 *Non permissive*

```
definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 mode μ of x1, x2, ..., xn -> Θ means :ident:
  Φ(x1, x2, ..., xn, it);
 existence;
end;

a is μ of x1, x2, ..., xn
proof
 thus Φ(x1, x2, ..., xn, a);
end;

definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 mode μ of x1, x2, ..., xn -> Θ means :ident:
  Φ1(x1, x2, ..., xn, it) if Γ1(x1, x2, ..., xn),
  Φ2(x1, x2, ..., xn, it) if Γ2(x1, x2, ..., xn),
  Φ3(x1, x2, ..., xn, it) if Γ3(x1, x2, ..., xn)
  otherwise Φn(x1, x2, ..., xn, it);
 existence;
```

```
 consistency;
end;
```

```
a is μ of x1, x2, ..., xn
proof
 per cases;
 case Γ1(x1, x2, ..., xn);
  thus Φ1(x1, x2, ..., xn, a);
 end;
 case Γ2(x1, x2, ..., xn);
  thus Φ2(x1, x2, ..., xn, a);
 end;
 case Γ3(x1, x2, ..., xn);
  thus Φ3(x1, x2, ..., xn, a);
 end;
 case not Γ1(x1, x2, ..., xn) & not Γ2(x1, x2, ..., xn) & not Γ3(x1, x2, ..., xn);
  thus Φn(x1, x2, ..., xn, a);
 end;
end;
```

A.7.2.2 *Permissive*

```
definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 assume Ψ(x1, x2, ..., xn);
 mode μ of x1, x2, ..., xn -> Θ means :ident:
  Φ(x1, x2, ..., xn, it);
 existence;
end;
```

```
a is μ of x1, x2, ..., xn
proof
 thus Ψ(x1, x2, ..., xn);
 thus Φ(x1, x2, ..., xn, a);
end;
```

```
definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 assume Ψ(x1, x2, ..., xn);
 mode μ of x1, x2, ..., xn -> Θ means :ident:
  Φ1(x1, x2, ..., xn, it) if Γ1(x1, x2, ..., xn),
  Φ2(x1, x2, ..., xn, it) if Γ2(x1, x2, ..., xn),
  Φ3(x1, x2, ..., xn, it) if Γ3(x1, x2, ..., xn)
  otherwise Φn(x1, x2, ..., xn, it);
 existence;
 consistency;
end;
```

```
a is μ of x1, x2, ..., xn
proof
```

```
 thus Ψ(x_1, x_2, ..., x_n);
 per cases;
 case Γ_1(x_1, x_2, ..., x_n);
  thus Φ_1(x_1, x_2, ..., x_n, a);
 end;
 case Γ_2(x_1, x_2, ..., x_n);
  thus Φ_2(x_1, x_2, ..., x_n, a);
 end;
 case Γ_3(x_1, x_2, ..., x_n);
  thus Φ_3(x_1, x_2, ..., x_n, a);
 end;
 case not Γ_1(x_1, x_2, ..., x_n) & not Γ_2(x_1, x_2, ..., x_n) & not Γ_3(x_1, x_2, ..., x_n);
  thus Φ_n(x_1, x_2, ..., x_n, a);
 end;
end;
```

A.7.3 *Attributes.*

A.7.3.1 *Non permissive*

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 attr x_n is α means :ident:
  Φ(x_1, x_2, ..., x_n);
end;
```

```
x_n is α
proof
 thus Φ(x_1, x_2, ..., x_n);
end;
```

```
definition
 let x_1 be Θ_1, x_2 be Θ_2, ..., x_n be Θ_n;
 attr x_n is α means :ident:
  Φ_1(x_1, x_2, ..., x_n) if Γ_1(x_1, x_2, ..., x_n),
  Φ_2(x_1, x_2, ..., x_n) if Γ_2(x_1, x_2, ..., x_n),
  Φ_3(x_1, x_2, ..., x_n) if Γ_3(x_1, x_2, ..., x_n)
  otherwise Φ_n(x_1, x_2, ..., x_n);
 consistency;
end;
```

```
x_n is α
proof
 per cases;
 case Γ_1(x_1, x_2, ..., x_n);
  thus Φ_1(x_1, x_2, ..., x_n);
 end;
 case Γ_2(x_1, x_2, ..., x_n);
  thus Φ_2(x_1, x_2, ..., x_n);
 end;
```

```
 case Γ3(x1, x2, ..., xn);
  thus Φ3(x1, x2, ..., xn);
 end;
 case not Γ1(x1, x2, ..., xn) & not Γ2(x1, x2, ..., xn) & not Γ3(x1, x2, ..., xn);
  thus Φn(x1, x2, ..., xn);
 end;
end;
```

A.7.3.2 *Permissive*

```
definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 assume Ψ(x1, x2, ..., xn);
 attr xn is α means :ident:
  Φ(x1, x2, ..., xn);
end;

xn is α
proof
 thus Ψ(x1, x2, ..., xn);
 thus Φ(x1, x2, ..., xn);
end;

definition
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 assume Ψ(x1, x2, ..., xn);
 attr xn is α means :ident:
  Φ1(x1, x2, ..., xn) if Γ1(x1, x2, ..., xn),
  Φ2(x1, x2, ..., xn) if Γ2(x1, x2, ..., xn),
  Φ3(x1, x2, ..., xn) if Γ3(x1, x2, ..., xn)
  otherwise Φn(x1, x2, ..., xn);
 consistency;
end;

xn is α
proof
 thus Ψ(x1, x2, ..., xn);
 per cases;
 case Γ1(x1, x2, ..., xn);
  thus Φ1(x1, x2, ..., xn);
 end;
 case Γ2(x1, x2, ..., xn);
  thus Φ2(x1, x2, ..., xn);
 end;
 case Γ3(x1, x2, ..., xn);
  thus Φ3(x1, x2, ..., xn);
 end;
 case not Γ1(x1, x2, ..., xn) & not Γ2(x1, x2, ..., xn) & not Γ3(x1, x2, ..., xn);
  thus Φn(x1, x2, ..., xn);
 end;
```

```
end;
```

A.8 Identify

A.8.1 *without* ***when*** *statement*

```
registration
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 identify τ1(x1,x2,...,xn) with τ2(x1,x2,...,xn);
 compatibility
 proof
  thus τ1(x1,x2,...,xn) = τ2(x1,x2,...,xn);
 end;
end;
```

A.8.2 *with* ***when*** *statement*

```
registration
 let x1 be Θ1, x2 be Θ2, ..., xn be Θn;
 let y1 be Ξ1, y2 be Ξ2, ..., yn be Ξn;
 identify τ1(x1,x2,...,xn) with τ2(y1,y2,...,yn)
  when x1 = y1, x2 = y2, ..., xn = yn;
 compatibility
 proof
  thus x1 = y1 & x2 = y2 & ... & xn = yn
   implies τ1(x1,x2,...,xn) = τ2(y1,y2,...,yn);
 end;
end;
```

B. THE SYNTAX OF THE MIZAR LANGUAGE

```
Mizar Language Syntax

Last modified: November 24, 2010

System terms:
      File-Name,
      Identifier,
      Numeral,
      Symbol.



********        Article

Article = Environment-Declaration Text-Proper .



********        Environment
```

```
Environment-Declaration = ‘‘environ’’ { Directive } .

Directive = Vocabulary-Directive | Library-Directive |
   Requirement-Directive .

Vocabulary-Directive = ‘‘vocabularies’’ Vocabulary-Name
   { ‘‘,’’ Vocabulary-Name } ‘‘;’’ .

Vocabulary-Name = File-Name .

Library-Directive =
 ( ‘‘notations’’ |
   ‘‘constructors’’ |
   ‘‘registrations’’ |
   ‘‘definitions’’ |
   ‘‘theorems’’ |
   ‘‘schemes’’ ) Article-Name { ‘‘,’’ Article-Name } ‘‘;’’ .

Article-Name = File-Name .

Requirement-Directive =
   ‘‘requirements’’ Requirement { ‘‘,’’ Requirement } ‘‘;’’ .

Requirement = File-Name .



********        Text Proper

Text-Proper = Section { Section } .

Section = ‘‘begin’’ { Text-Item } .

Text-Item =
   Reservation | Definitional-Item | Registration-Item |
   Notation-Item | Theorem | Scheme-Item | Auxiliary-Item |
   Canceled-Theorem .

Reservation = ‘‘reserve’’ Reservation-Segment
   { ‘‘,’’ Reservation-Segment } ‘‘;’’ .

Reservation-Segment = Reserved-Identifiers ‘‘for’’ Type-Expression .

Reserved-Identifiers = Identifier { ‘‘,’’ Identifier } .

Definitional-Item = Definitional-Block ‘‘;’’ .
```

```
Registration-Item = Registration-Block ‘‘;’’ .

Notation-Item = Notation-Block ‘‘;’’ .

Definitional-Block = ‘‘definition’’ { Definition-Item | Definition }
   [ Redefinition-Block ] ‘‘end’’ .

Redefinition-Block = ‘‘redefine’’ { Definition-Item | Definition } .

Registration-Block = ‘‘registration’’
   { Loci-Declaration |  Cluster-Registration |
     Identify-Registration |  Canceled-Registration }
   ‘‘end’’ .

Notation-Block =
   ‘‘notation’’ { Loci-Declaration | Notation-Declaration }
   ‘‘end’’ .

Definition-Item =
   Loci-Declaration | Permissive-Assumption | Auxiliary-Item .

Notation-Declaration = Attribute-Synonym | Attribute-Antonym |
   Functor-Synonym | Mode-Synonym | Predicate-Synonym |
   Predicate-Antonym .

Loci-Declaration =
   ‘‘let’’ Qualified-Variables [ ‘‘such’’ Conditions ] ‘‘;’’ .

Permissive-Assumption = Assumption .

Definition = Structure-Definition | Mode-Definition |
   Functor-Definition | Predicate-Definition |
   Attribute-Definition | Canceled-Definition .

Structure-Definition = ‘‘struct’’ [ ‘‘(‘‘ Ancestors ‘‘)’’ ]
   Structure-Symbol [ ‘‘over’’ Loci ]
   ‘‘(#’’ Fields ‘‘#)’’ ‘‘;’’ .

Ancestors =
   Structure-Type-Expression { ‘‘,’’ Structure-Type-Expression } .

Structure-Symbol = Symbol .

Loci = Locus { ‘‘,’’ Locus } .

Fields = Field-Segment { ‘‘,’’ Field-Segment } .
```

```
Locus = Variable-Identifier .

Variable-Identifier = Identifier .

Field-Segment =
   Selector-Symbol { ‘‘,’’ Selector-Symbol } Specification .

Selector-Symbol = Symbol .

Specification = ‘‘->’’ Type-Expression .

Mode-Definition = ‘‘mode’’ Mode-Pattern
   ( [ Specification ] [ ‘‘means’’ Definiens ] ‘‘;’’
       Correctness-Conditions |
   ‘‘is’’ Type-Expression ‘‘;’’ ) .

Mode-Pattern = Mode-Symbol [ ‘‘of’’ Loci ] .

Mode-Symbol = Symbol | ‘‘set’’ .

Mode-Synonym = ‘‘synonym’’ Mode-Pattern ‘‘for’’ Mode-Pattern ‘‘;’’ .

Definiens = Simple-Definiens | Conditional-Definiens .

Simple-Definiens =
   [ ‘‘:’’ Label-Identifier ‘‘:’’ ] ( Sentence | Term-Expression ) .

Label-Identifier = Identifier .

Conditional-Definiens = [ ‘‘:’’ Label-Identifier ‘‘:’’ ]
   Partial-Definiens-List
   [ ‘‘otherwise’’ ( Sentence | Term-Expression ) ] .

Partial-Definiens-List =
    Partial-Definiens { ‘‘,’’ Partial-Definiens } .

Partial-Definiens = ( Sentence | Term-Expression ) ‘‘if’’ Sentence .

Functor-Definition = ‘‘func’’ Functor-Pattern [ Specification ]
   [ ( ‘‘means’’ | ‘‘equals’’ ) Definiens ] ‘‘;’’
   Correctness-Conditions { Functor-Property } .

Functor-Pattern = [ Functor-Loci ] Functor-Symbol [ Functor-Loci ] |
   Left-Functor-Bracket Loci Right-Functor-Bracket .

Functor-Property = ( ‘‘commutativity’’ | ‘‘idempotence’’ |
```

```
   ‘‘involutiveness’’ | ‘‘projectivity’’ )
   Justification ‘‘;’’ .

Functor-Synonym =
   ‘‘synonym’’ Functor-Pattern ‘‘for’’ Functor-Pattern ‘‘;’’ .

Functor-Loci = Locus | ‘‘(‘‘ Loci ‘‘)’’ .

Functor-Symbol = Symbol .

Left-Functor-Bracket = Symbol | ‘‘{‘‘ | ‘‘[‘‘ .

Right-Functor-Bracket = Symbol | ‘‘}’’ | ‘‘]’’ .

Predicate-Definition =
   ‘‘pred’’ Predicate-Pattern [ ‘‘means’’ Definiens ] ‘‘;’’
   Correctness-Conditions { Predicate-Property } .

Predicate-Pattern = [ Loci ] Predicate-Symbol [ Loci ] .

Predicate-Property = ( ‘‘symmetry’’ | ‘‘asymmetry’’ |
   ‘‘connectedness’’ | ‘‘reflexivity’’ | ‘‘irreflexivity’’ )
   Justification ‘‘;’’ .

Predicate-Synonym =
   ‘‘synonym’’ Predicate-Pattern ‘‘for’’ Predicate-Pattern ‘‘;’’ .

Predicate-Antonym =
   ‘‘antonym’’ Predicate-Pattern ‘‘for’’ Predicate-Pattern ‘‘;’’ .

Predicate-Symbol = Symbol | ‘‘=’’ .

Attribute-Definition =
   ‘‘attr’’ Attribute-Pattern ‘‘means’’ Definiens ‘‘;’’
   Correctness-Conditions .

Attribute-Pattern =
   Locus ‘‘is’’ [ Attribute-Loci ] Attribute-Symbol .

Attribute-Synonym =
   ‘‘synonym’’ Attribute-Pattern ‘‘for’’ Attribute-Pattern ‘‘;’’ .

Attribute-Antonym =
   ‘‘antonym’’ Attribute-Pattern ‘‘for’’ Attribute-Pattern ‘‘;’’ .

Attribute-Symbol = Symbol .
```

```
Attribute-Loci = Loci | ‘‘(‘‘ Loci ‘‘)’’ .

Canceled-Definition = ‘‘canceled’’ [ Numeral ] ‘‘;’’ .

Canceled-Registration = ‘‘canceled’’ [ Numeral ] ‘‘;’’ .

Cluster-Registration = Existential-Registration |
   Conditional-Registration |
   Functorial-Registration .

Existential-Registration =
   ‘‘cluster’’ Adjective-Cluster Type-Expression ‘‘;’’
   Correctness-Conditions .

Adjective-Cluster = { Adjective } .

Adjective = [ ‘‘non’’ ] [ Adjective-Arguments ] Attribute-Symbol .

Conditional-Registration = ‘‘cluster’’ Adjective-Cluster ‘‘->’’
   Adjective-Cluster Type-Expression ‘‘;’’
   Correctness-Conditions .

Functorial-Registration = ‘‘cluster’’ Term-Expression ‘‘->’’
   Adjective-Cluster [ Type-Expression ] ‘‘;’’
   Correctness-Conditions .

Identify-Registration =
   ‘‘identify’’ Functor-Pattern ‘‘with’’ Functor-Pattern
   [ ‘‘when’’ Locus ‘‘=’’ Locus { ‘‘,’’ Locus ‘‘=’’ Locus } ] ‘‘;’’
   Correctness-Conditions .

Correctness-Conditions = { Correctness-Condition }
   [ ‘‘correctness’’ Justification ‘‘;’’ ] .

Correctness-Condition = ( ‘‘existence’’ | ‘‘uniqueness’’ |
   ‘‘coherence’’ | ‘‘compatibility’’ | ‘‘consistency’’ )
   Justification ‘‘;’’ .

Theorem = ‘‘theorem’’ Compact-Statement .

Scheme-Item = Scheme-Block ‘‘;’’ .

Scheme-Block = ‘‘scheme’’ Scheme-Identifier
   ‘‘{‘‘ Scheme-Parameters ‘‘}’’ ‘‘:’’ Scheme-Conclusion
   [ ‘‘provided’’ Scheme-Premise { ‘‘and’’ Scheme-Premise } ]
   Reasoning ‘‘end’’ .
```

```
Scheme-Identifier = Identifier .

Scheme-Parameters = Scheme-Segment { ‘‘,’’ Scheme-Segment } .

Scheme-Conclusion = Sentence .

Scheme-Premise = Proposition .

Scheme-Segment = Predicate-Segment | Functor-Segment .

Predicate-Segment =
   Predicate-Identifier { ‘‘,’’ Predicate-Identifier }
   ‘‘[‘‘ [ Type-Expression-List ] ‘‘]’’ .

Predicate-Identifier = Identifier .

Functor-Segment = Functor-Identifier { ‘‘,’’ Functor-Identifier }
   ‘‘(‘‘ [ Type-Expression-List ] ‘‘)’’ Specification .

Functor-Identifier = Identifier .

Auxiliary-Item = Statement | Private-Definition .

Canceled-Theorem = ‘‘canceled’’ [ Numeral ] ‘‘;’’ .

Private-Definition = Constant-Definition |
   Private-Functor-Definition |
   Private-Predicate-Definition .

Constant-Definition = ‘‘set’’ Equating-List ‘‘;’’ .

Equating-List = Equating { ‘‘,’’ Equating } .

Equating = Variable-Identifier ‘‘=’’ Term-Expression .

Private-Functor-Definition =
   ‘‘deffunc’’ Private-Functor-Pattern ‘‘=’’ Term-Expression .

Private-Predicate-Definition =
   ‘‘defpred’’ Private-Predicate-Pattern ‘‘means’’ Sentence .

Private-Functor-Pattern =
   Functor-Identifier ‘‘(‘‘ [ Type-Expression-List ] ‘‘)’’ .

Private-Predicate-Pattern =
   Predicate-Identifier ‘‘[‘‘ [ Type-Expression-List ] ‘‘]’’ .
```

```
Reasoning = { Reasoning-Item }
   [ ‘‘per’’ ‘‘cases’’ Simple-Justification ‘‘;’’
     ( Case-List | Suppose-List ) ] .

Case-List = Case { Case } .

Case = ‘‘case’’ ( Proposition | Conditions ) ‘‘;’’
   Reasoning ‘‘end’’ ‘‘;’’ .

Suppose-List = Suppose { Suppose } .

Suppose = ‘‘suppose’’ ( Proposition | Conditions ) ‘‘;’’
   Reasoning ‘‘end’’ ‘‘;’’ .

Reasoning-Item = Auxiliary-Item | Skeleton-Item .

Skeleton-Item = Generalization | Assumption |
   Conclusion | Exemplification .

Generalization = ‘‘let’’ Qualified-Variables
   [ ‘‘such’’ Conditions ] ‘‘;’’ .

Assumption = Single-Assumption | Collective-Assumption |
   Existential-Assumption .

Single-Assumption = ‘‘assume’’ Proposition ‘‘;’’ .

Collective-Assumption = ‘‘assume’’ Conditions ‘‘;’’ .

Existential-Assumption = ‘‘given’’ Qualified-Variables
   ‘‘such’’ Conditions ‘‘;’’ .

Conclusion = ( ‘‘thus’’ | ‘‘hence’’ ) Compact-Statement |
   Diffuse-Conclusion .

Diffuse-Conclusion = ‘‘thus’’ Diffuse-Statement |
   ‘‘hereby’’ Reasoning ‘‘end’’ ‘‘;’’ .

Exemplification = ‘‘take’’ Example { ‘‘,’’ Example } ‘‘;’’ .

Example = Term-Expression |
   Variable-Identifier ‘‘=’’ Term-Expression .

Statement = [ ‘‘then’’ ] Linkable-Statement | Diffuse-Statement .

Linkable-Statement = Compact-Statement | Choice-Statement |
   Type-Changing-Statement | Iterative-Equality .
```

```
Compact-Statement = Proposition Justification ‘‘;’’ .

Choice-Statement = ‘‘consider’’ Qualified-Variables
   ‘‘such’’ Conditions Simple-Justification ‘‘;’’ .

Type-Changing-Statement = ‘‘reconsider’’ Type-Change-List
   ‘‘as’’ Type-Expression Simple-Justification ‘‘;’’ .

Type-Change-List = ( Equating | Variable-Identifier )
   { ‘‘,’’ ( Equating | Variable-Identifier ) } .

Iterative-Equality = [ Label-Identifier ‘‘:’’ ]
   Term-Expression ‘‘=’’ Term-Expression Simple-Justification
   ‘‘.=’’ Term-Expression Simple-Justification
 { ‘‘.=’’ Term-Expression Simple-Justification } ‘‘;’’ .

Diffuse-Statement = [ Label-Identifier ‘‘:’’ ]
   ‘‘now’’ Reasoning ‘‘end’’ ‘‘;’’ .

Justification = Simple-Justification | Proof .

Simple-Justification = Straightforward-Justification |
   Scheme-Justification .

Proof = ( ‘‘proof’’ | ‘‘@proof’’ ) Reasoning ‘‘end’’ .

Straightforward-Justification = [ ‘‘by’’ References ] .

Scheme-Justification =
   ‘‘from’’ Scheme-Reference [ ‘‘(‘‘ References ‘‘)’’ ] .

References = Reference { ‘‘,’’ Reference } .

Reference = Local-Reference | Library-Reference .

Scheme-Reference = Local-Scheme-Reference |
   Library-Scheme-Reference .

Local-Reference = Label-Identifier .

Local-Scheme-Reference = Scheme-Identifier .

Library-Reference = Article-Name ‘‘:’’
   ( Theorem-Number | ‘‘def’’ Definition-Number )
   { ‘‘,’’ ( Theorem-Number | ‘‘def’’ Definition-Number ) } .
```

```
Library-Scheme-Reference =
   Article-Name ‘‘:’’ ‘‘sch’’ Scheme-Number .

Theorem-Number = Numeral .

Definition-Number = Numeral .

Scheme-Number = Numeral .

Conditions = ‘‘that’’ Proposition { ‘‘and’’ Proposition } .

Proposition = [ Label-Identifier ‘‘:’’ ] Sentence .

Sentence = Formula-Expression .



********        Expressions

Formula-Expression = ‘‘(‘‘ Formula-Expression ‘‘)’’ |
   Atomic-Formula-Expression |
   Quantified-Formula-Expression |
   Formula-Expression ‘‘&’’ Formula-Expression |
   Formula-Expression ‘‘or’’ Formula-Expression |
   Formula-Expression ‘‘implies’’ Formula-Expression |
   Formula-Expression ‘‘iff’’ Formula-Expression |
   ‘‘not’’ Formula-Expression |
   ‘‘contradiction’’ |
   ‘‘thesis’’ .

Atomic-Formula-Expression =
   [ Term-Expression-List ] Predicate-Symbol
      [ Term-Expression-List ] |
   Predicate-Identifier ‘‘[‘‘ [ Term-Expression-List ] ‘‘]’’ |
   Term-Expression ‘‘is’’ Adjective { Adjective } |
   Term-Expression ‘‘is’’ Type-Expression .

Quantified-Formula-Expression =
   ‘‘for’’ Qualified-Variables [ ‘‘st’’ Formula-Expression ]
      ( ‘‘holds’’ Formula-Expression |
                  Quantified-Formula-Expression ) |
   ‘‘ex’’ Qualified-Variables ‘‘st’’ Formula-Expression .

Qualified-Variables = Implicitly-Qualified-Variables |
   Explicitly-Qualified-Variables |
   Explicitly-Qualified-Variables ‘‘,’’
      Implicitly-Qualified-Variables .
```

```
Implicitly-Qualified-Variables = Variables .

Explicitly-Qualified-Variables =
   Qualified-Segment { ``,'' Qualified-Segment } .

Qualified-Segment = Variables Qualification .

Variables = Variable-Identifier { ``,'' Variable-Identifier } .

Qualification = ( ``being'' | ``be'' ) Type-Expression .

Type-Expression = ``(`` Type-Expression ``)'' |
   Adjective-Cluster Type-Expression |
   Radix-Type .

Structure-Type-Expression = ``(`` Structure-Type-Expression ``)'' |
   Adjective-Cluster Structure-Symbol
      [ ``over'' Term-Expression-List ] .

Radix-Type = Mode-Symbol [ ``of'' Term-Expression-List ] |
   Structure-Symbol [ ``over'' Term-Expression-List ] .

Type-Expression-List = Type-Expression { ``,'' Type-Expression } .

Term-Expression = ``(`` Term-Expression ``)'' |
   [ Arguments ] Functor-Symbol [ Arguments ] |
   Left-Functor-Bracket Term-Expression-List Right-Functor-Bracket |
   Functor-Identifier ``(`` [ Term-Expression-List ] ``)'' |
   Structure-Symbol ``(#'' Term-Expression-List ``#)'' |
   Variable-Identifier |
   ``{`` Term-Expression [ Postqualification ] ``:''
         Sentence ``}'' |
   Numeral |
   Term-Expression ``qua'' Type-Expression |
   ``the'' Selector-Symbol ``of'' Term-Expression |
   ``the'' Selector-Symbol |
   ``the'' Type-Expression |
   Private-Definition-Parameter |
   ``it'' .

Arguments = Term-Expression | ``(`` Term-Expression-List ``)'' .

Adjective-Arguments = Term-Expression-List |
   ``(`` Term-Expression-List ``)'' .

Term-Expression-List = Term-Expression { ``,'' Term-Expression } .
```

```
Postqualification = ‘‘where’’ Postqualifying-Segment
   { ‘‘,’’ Postqualifying-Segment } .

Postqualifying-Segment = Postqualified-Variable
   { ‘‘,’’ Postqualified-Variable } ‘‘is’’ Type-Expression .

Postqualified-Variable = Identifier .

Private-Definition-Parameter = ‘‘$1’’ | ‘‘$2’’ | ‘‘$3’’ | ‘‘$4’’ |
   ‘‘$5’’ | ‘‘$6’’ | ‘‘$7’’ | ‘‘$8’’ | ‘‘$9’’ | ‘‘$10’’ .
```

ACKNOWLEDGMENT

The authors wish to express their sincere appreciation to the outstanding work of Andrzej Trybulec, the inventor of Mizar and the leader of the Mizar project.

References

[1] G. Bancerek and P. Rudnicki. Information retrieval in MML. In *MKM'03: Proceedings of the Second International Conference on Mathematical Knowledge Management*, pp. 119–132, 2003.

[2] P. Corbineau. A declarative language for the Coq proof assistant. In *Types for Proofs and Programs*, LNCS 4941, pp. 69–84, 2008.

[3] F. B. Fitch. *Symbolic Logic. An Introduction.* The Ronald Press Company, 1952.

[4] A. Grabowski and A. Naumowicz. Computer Reconstruction of the Body of Mathematics *Studies in Logic, Grammar and Rhetoric*, 2009.

[5] J. Harrison. A Mizar Mode for HOL. In *TPHOLs'96: Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics*, pp. 203–220, 1996.

[6] S. Jaśkowski. On the rules of supposition in formal logic. *Studia Logica*, 1, 1934.

[7] A. Korniłowicz. How to define terms in Mizar effectively. In [4].

[8] R. Matuszewski and P. Rudnicki. Mizar: the first 30 years. *Mechanized Mathematics and Its Applications*, 4(1), pp. 3–24, 2005.

[9] R. Matuszewski and A. Zalewska. From Insight to Proof. Festschrift in Honour of Andrzej Trybulec. *Studies in Logic, Grammar and Rhetoric*, 2007.

[10] Mizar home page: `http://mizar.org`.

[11] A. Naumowicz. Enhanced processing of adjectives in Mizar. In [4].

[12] A. Naumowicz. Teaching How to Write a Proof. In *Formed 2008: Formal Methods in Computer Science Education*, pp. 91–100, 2008.

[13] A. Naumowicz and C. Byliński. Improving Mizar texts with properties and requirements. In A. Asperti, editor, *MKM-2004*, LNCS 3119, pp. 290–301, 2004.

[14] A. Naumowicz and A. Korniłowicz. A Brief Overview of Mizar. In *S. Berghofer et al. (Eds.), TPHOLs 2009*, LNCS 5674, Springer-Verlag Berlin Heidelberg, 2009.

[15] K. Ono. On a practical way of describing formal deductions. *Nagoya Mathematical Journal*, 21, 1962.

[16] QED Manifesto: `http://www.rbjones.com/rbjpub/logic/qedres00.htm`.

[17] Ch. Schwarzweller Mizar attributes. A technique to encode mathematical knowledge. In [9].

[18] D. Syme. Three tactic theorem proving. In *TPHOLs'99: Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics*, pp. 203–220, 1999.

[19] A. Trybulec. Some Features of the Mizar Language. In Proceedings of ESPRIT Workshop, Torino 1993.

[20] A. Trybulec. Tarski Grothendieck set theory. *Formalized Mathematics*, 1(1), pp. 9–11, 1990.

[21] J. Urban. XML-izing Mizar: Making Semantic Processing and Presentation of MML Easy. In *Mathematical Knowledge Management: MKM 2005*, LNCS 3863, pp. 346–360, 2006.

[22] M. Wenzel and F. Wiedijk. A comparison of Mizar and Isar. *Journal of Automated Reasoning*, 29(3-4), pp. 389–411, 2002.

[23] F. Wiedijk. Formal Proof Sketches. In *Types for Proofs and Programs: TYPES 2003*, LNCS 3085, pp. 378–393, 2004.

[24] F. Wiedijk. Mizar Light for HOL Light. In *Theorem Proving in Higher Order Logics: TPHOLs 2001*, LNCS 2152, pp. 378–393, 2001.