Mizar 言語の特徴

(Some Features of the Mizar Language)

Andrzej Trybulec
Institute of Mathematics
Warsaw University in Białystok
15-267 Białystok, Akademicka 2, Poland
Translation by l'Hospitalier

2010
日本語版（1.0a 校）

前文 (preface)　Mizar 言語を短い論文で完全に提示するのは不可能です。　興味を引きそうなトピックをいくつか選択するか、あるいは我々の研究の考察（insight）を提示しましょう。 この提示の大部分を字句解析（tokenization）、構文の識別、それに類似の事柄に関連する技術的問題に割きました。　それらは極めてしばしば、不公平にも無視されているのです。　それらの問題の正しい解決はシステムの実用的な価値に大きなインパクトをもちます。　コメントを始めるのに、どの文字（character）を使うかについての原理（law）（そしてこの原理を強化することを可能にする国際的機関について）から始めることにします。

私は Mizar コミュニティーで使われるいろいろな略語にあまりにも慣れすぎているので、説明なしにそれを使ってしまうのではないかが心配です。　SUM（ポーランド語：Stowarzyszenie Użytkowników Mizara, 逐語的に：Association of Mizar Users）は小規模の科学者の集まりで（約４０人、４つのサークル、うち３つはポーランド、１つは日本）、ポーランドに本拠を置き、通常英語で　－the Mizar Society－　と呼ばれます。　SUM の主な活動は Mizar article の収集です。　SUM のライブラリ委員会は article のアクセプトと編集に責任があります。　アクセプトされた article は MML　－the Main Mizar Library を構成します。　MizUG　－Mizar Users Group はベルギーに本拠を置く組織です。　これは Mizar システムの配布と Formalized Mathematics の出版に責任があります。

本文を読みやすくするためのルールを採用しました。　Mizar article の断片は **`Courier`** フォントで。　これには単独の Mizar シンボルも含みます、例えば **`set`**。　イタリック体は Mizar の文法的な構文の名前に使用、もしそれらが骨格となる（skeletal）Mizar text に使われた場合、Mizar 予約語は太字で、それ以外はロマン体を使用する。

内容（Contents）

1　序文（Introduction）

Mizar article は２つの部分からなります：　*環境部の宣言*と*本文*です。　*環境部の宣言*は **environ** で始まり*指令（Directives）*で構成されます。　*本文（Text Proper）*は*セクション（Section）*の有限の一続きで、全ての*セクション*は **begin** で始まる*事項（Items）*からなります。　本文をセクションに分割することは編集目的のためだけに行われ、article の正しさには影響がありません。

environ
　Directive
　. . .
　Directive
begin
　Item
　…
　Item
…
begin
　Item
　…
　Item

Article の２つの部分は２つの別のプログラムで処理されます。　アコモデータ(//??Accomodator->Accommodator)とベリファイヤ（Verifier）です。　アコモデータ(//??Accomodator->Accommodator)は*環境部の宣言*を処理し、*指令（Directives）*で要求されてデータベースからインポートされた情報がしまってある多くのファイルから構成される環境（the Environment）を作成します。

Verifier は直接データベースと連絡することなく環境に格納された情報だけを使って*本文（Text Proper）*の正しさを検証します。　*指令（Directives）*は２種類あります：*Vocabulary Directive* と *Library Directive* です。　*Vocabulary Directive* は Vocabulary からシンボルを要求します。　Vocabulary というのはシンボルが定義されている ASCII ファイルです。　*Vocabulary Directive* は：

v**ocabulary** *Vocabulary 名, …, Vocabulary 名;*

という形をとります。　*Library Directives* はデータベースから特定の情報を要求します。　都合の悪いことに（unfortunately）われわれはちょうどデータベースの変更を開始したところです。　article のなかで使用される概念の枠組み（conceptual framework）を要求する Directives は現時点で

signature　*Article Name, … Article Name;*

と書かれますが、将来は２つに分割されて

notations　*Article Name, … Article Name;*
constructors　*Article Name, … Article Name;*

となるでしょう。
この理由は Mizar においては、もしオリジナルの記法が気に入らなければ、同義語（synonyms）の（そして意味があれば反義語（antonym）も）導入が許されているからです。　それゆえ構文（constructor）（述語やファンクタなど）はいろいろな異なった方法で表現されます。　**constructors** という指令は概念の枠組み（conceptual framework）を作成するのに使用されるべきアコモデータ（//??Accomodator-> Accommodator）をわれわれの article に通知します、**notations** 指令はどの article から記法（notation）が取り込まれるべきかを通知するでしょう。
　残っている指令は：

clusters *Article Name, … , Article Name;*

（これの意味は後ほど説明します）そして

definitions *Article Name, … , Article Name;*

もし定義の拡張（definitional expansion）による証明で使おうとするなら、definientia*（//??->definienda:定義されるべきもの(pl.)？）を要求します。
Mizar では

```
A c= B
```

proof let x be Any; assume x ∈ A; ...; hence x ∈ B; end;

そのような証明を可能にするため Verifier は包含（inclusion）の定義する項（definiens）を知らなければなりません。

theorems *Article Name, . . . , Article Name;*

これはリストアップされた定理への参照を可能にします。　そのような参照は *Straightforward Justification* でもちいられ、それは

by *Reference, . . . , Reference;*

という形をしています。　*Reference* というのは article の中の statement の参照である *Private Reference*、あるいはデータベースに格納された定理（theorem）である *Library Reference* のどちらかです。　それは２つの型：*Article name: Theorem Number* か、あるいは *Article name:* **def** *Definition Number* をもちます。　２番目のものはいわゆる定義定理（definitional theorem）、定義の意味を記述する自動的に作成される定義、を参照します。　これはちょっとした混乱の原因になります。　ある article で定義定理にアクセスするためには、その名前を **definition** 指令ではなく **theorem** 指令に挿入しなければなりません。

scheme *Article Name, . . . , Article Name;*

これは scheme、それはそのなかで２階の自由変数が現れる定理のことですが、にアクセスするのに使われます。　これの良い例は帰納法の scheme[1]です。この scheme で“for every natural number the property　α　holds”をジャスティファイするには、まず初めに

A:　$\alpha(0)$;
. . .
B: **for** k **being** Nat **st**　$\alpha(k)$ **holds** α(k+1)
. . .

それから

for k **being** Nat **holds** $\alpha(k)$ from Ind*(A,B);*

とします。

次に考えるプログラムは Extractor です。　それは *Article* からパブリックの情報（public information）を抽出し（プライベートな情報、例えばジャスティフィケーションや補題は無視されます）、それをライブラリ ファイルに格納します。　それらのファイルは、次の article のための環境（Environment）を作

成するためにアコモデータ（//??Accomodator ->Accommodator）によって使用されます。

データベースは：パブリックデータベースとプライベートデータベースの２つの部分から構成されます。　Mizar Society から配布されるパブリックデータベースは Main Mizar Library(MML)へ投稿された article を使って作成されたライブラリファイルから構成されます、プライベートはユーザ（あるいはユーザのグループ)によりプライベートライブラリに格納された article に Extractor を使用して作成されるでしょう。　しかし、パブリックデータベースに移管されたライブラリファイルは最適化されます。　大きなプライベートライブラリの使用はトラブルを起こすかもしれないし、作者がそのライブラリの質に確信が持てたなら直ちにそれを MML に投稿するのは良い考え方（policy）です（という理由で（訳注：ライブラリの質を確かめるため）それを使って２つの article を書くのは通常のことです）。

本文（Text Proper）の記述についてあまり詳細にわたるのは好ましくありません。　後のほうで使う構文に名前が出てくるものについての記述にとどめるようにします。

*ブロック（Block）*は*事項（Items）*からなります。　それらには導入部となる*予約語（Reserved Word）*があり、ブロックが異なると予約語も異なります。　いくつかは **end** で終わります。　*プルーフ（Proof）*は１つ **end** を持ちます、*セクション（Section）*は持ちません。　*定義ブロック（Definition block）*というのは定義を導入するためのブロックです。　その文法は：

definition　*Definitional Item, . . ., Definitional Item*　**end**;

となります。

*事項（Items）*は常にセミコロンで終了するので、容易にそれとわかります。もっと言うとすべてのセミコロンは*事項*を終わらせます。　異なった*ブロック*では異なった*事項*のセットが使われます。　*セクション*においては*テキスト項（Text Item）*が、*定義（Definition）*では　― *定義項（Definitional Item）*が許されます。　*定義*は*定義ブロック*とその傍のセミコロンからなります。　*定義ブロック（Definitional Block）*はネストすることができません、それは *Text Item* であって、*Definitional Item* ではありません。　*構造体定義（Structure Definition）*は*テキスト項*と*定義項*の両方です。　その文法は：

struct*(Ancestors)*
　構造体のシンボル(Structure Symbol)　**over** *Loci*
　　≪*Field Segment, . . ., Field Segment*≫

2 つの部分：　*(Ancestors)*と **over** *Loci* はオプションであり省略可能です。

実際、構造体定義がテキスト項として使用されるなら、**over** *Loci* を省略せねばなりません。 *フィールドセグメントはセレクタシンボル (Selector Symbol)*、*…*、*セレクタシンボル -> 型表現（Type Expression）*
となります。

2 語彙目録（用語集）（Lexicon）

異なる article の用語（Lexicons）は異なるかも知れません。 用語法は語彙的な文脈 (lexical context)、すなわち*ボキャブラリ指令 (Vocabulary Directive)* におけるボキャブラリの要求に依存します。 それは以下に示す表象（token）のセットから構成されます：

予約語（Reserved Words）

aggregate and antonym as assume attr be begin being by canceled case cases cluster coherence consistency compatibility consider contradiction correctness def define definition deffunc defpred end environ ex exactly existence for from func given hence hereby holds if iff implies is it let means mode non not now of or otherwise over per pred prefix proof provided qua reconsider redefine reserve scheme selector set st struct such suppose synonym take that the then theorem thesis thus uniqueness where
& ( ) , : ; = [ ] { } ≪ (# ≫ #) .= ->
$1 $2 $3 $4 $5 $6 $7 $8
@proof

*語彙素（///??Lexems->Lexemes）*と表象（token）とは区別します。 ボキャブラリには類似（synonymous）の表象：

be being
≪ (#
≫ #)
func deffunc
pred defpred

があり (最後の２つはある文脈でのみ) 同じ*語彙素 (///??Lexems->Lexemes)*、同音異義の表象（homonymous token）を表します、そして異なる文脈では別の*語彙素（///??Lexems->Lexemes）*を表します。 それらは：

func pred set = { } { }

で、最初の２つを除いて、シンボルとして使われることもあります。　いくつかの表象は将来の使用ために予約されていて現在は全く使用しません。　それらは：

define selector aggregate prefix

です。　いくつかの語彙素は MML に投稿する article のなかで使用することはできません。　それらは：

canceled @proof

です。　**canceled** は MML の article から除去された定理を置き換えるのに使用され定理の正しい番号付けを維持します、そして@proof はプルーフの検証を一時的に抑止します（この機構は Zbigniew Karno により提案され article の検証をスピードアップするためにだけ使用されます）。

属性の名前（Property Names）

symmetry reflexivity irreflexivity
associativity transitivity commutativity

今までのところでは初めの３つしかサポートされていません。　実装（implementation）はもっと良くなることができると思います。

シンボル（Symbols）

前に説明したように、それらはボキャブラリで与えられます。　その名前がボキャブラリ指令、あるいは HIDDEN ボキャブラリあるいはそれの同音異義語（homonyms）にリストアップされているものだけが article の語彙目録にふくまれます。　ボキャブラリの各行はそのシンボルの種類を決める修飾子（qualifier）の１文字で始まり（述語のシンボル、ファンクタのシンボル等）、ついでシンボルの表現（representation）が直後に続きます。以下のシンボル：

R　*Predicate Symbol*
O　*Functor Symbol*
M　*Mode Symbol*
G　*Structure Symbol*
U　*Selector Symbol*
V　*Attribute Symbol*

K　*Left Functor Bracket*
L　*Right Functor Bracket*

が使用されます。　ファンクタのシンボルにはボキャブラリで追加の情報があたえられます： 数字で 0 から 255 のあいだの優先度（priority）です（指定がなければ 64）。　同音異義語（homonyms）のシンボルは（予約語の homonyms）はベリファイヤで認識されボキャブラリには有りません。それらは以下のように：

Set　これはモードのシンボル
=　これは述語のシンボル
{ }　これらはファンクタのカッコ
[ ]　これらもまたファンクタのカッコ

となります。　不必要な複雑さにおどろくかも知れません。　少なくともその一部は Mizar の進化や、異なる作者の異なる嗜好の結果です。

数詞（Numerals）

0 は数詞で 0 でない数字から始まる数字のひと続きは数詞です。

Article の名前（Article Names）

定理指令（**theorems** directive）のなかの article の名前だけが article の語彙目録に挿入されます。

スキームの名前（Scheme Names）

これらは識別子（identifiers）の形式をもちます。　それらがプライベートでないという事実はちょっと規則外（somewhat irregular）です。　将来これはどうにかする必要があるでしょう。　Czesław Byliński はそれらがライブラリ リファレンスの一種類として参照されるべきであるという考えを支持しています。　名前が schemes 指令のなかにリストアップされている article で導入された Scheme 名だけがアクセス可能です。

識別子（identifiers）

いかなる先行するシンボルのクラスに属さない文字の並び（Mizar は大文字小文字を区別します）、数字、アポトロフィ、そしてアンダースコア。（訳注：動詞がない文）　どうぞ*識別子*の概念は使用中の環境に依存するのに注意してください。　ライブラリ委員会のポリシーは*識別子*のように見えるシンボルは少なくとも 3 文字以上であるべきというものです。　都合の悪いことにファンクタシンボルの **U** が 2 つの集合の和集合として導入されました（それゆえ大多数の article で識別子としては使用できません）、**Ü**

（HIDDEN ボキャブラリで導入されている）は全く使用されませんが将来 **U** を置き換えると考えられています。　上で述べたポリシーによる配列（policy geometry）を実装するまえに、人々は betweenness* と共直線性（collinearity）を表す 1 文字の*述語シンボル（Predicate Symbols）*を導入するのに代替わりしました、われわれはそれらのシンボルを変更するよう迫る（press）つもりです。

（*訳注：ヒルベルトの axiom of betweenness のことか？）

不可視のキャラクタ（LF CR SP）は表象（token）の内側では使用できません。　しかし、表象を不可視キャラクタで分けることは不要です。　それらが分けられていないと、表象にスライスされるようなキャラクタの一続きを生成します。　規則は単純で、スライサー（それを行う手続き）は可能なかぎり長い表象を得ようと試み、それを刻んでいきます。　それから後戻りなしにその過程がくりかえされます。　もしそのセグメントの（訳注：切り出された）接頭辞（prefix）のどれもが表象としての基準を満たさないと、字句解析エラー（tokenization error）が（“Unkown token”のメッセージで）報告され、1 文字が切り離されます。

その手のスライスは時々問題をひき起こします。　例えば（E.g.）包含（inclusion）を表す表象に **c=** が使われています。　そこで

```
a ∩ c U c= c
```

と書くと文法エラーが発生すると思います、なぜなら **c=** は包含のシンボルとして取り扱われるからです。　対処法は簡単です。　ちょっとスペースを入れれば良い。

```
a ∩ c U c = c
```

経験を積んだユーザーにとっても、依然としてしばしば混乱しやすいのです。

私が、一見したところどうでもよい事柄に多くの文を割くのをいぶかしんでおられるかも知れません。　われわれの語彙目録（lexicon）に関しての問題は実用的で進歩するシステムで、それは便利で柔軟なものでなければならないと強く信じています。　これらの問題はより洗練されたレベルでも出現します。そのようなレベルだけでそれらを説明するのは難しくなります。　私は字句解析のレベルをわれわれが戦わねばならない問題の説明として使いたいのです。

3　記数法（Notation）

そのフォーマット（format）には構文シンボル（Constructor Symbol）がいくつ引数を使用するかが記載されています。　アコモデータ（//??Accomodator->

Accommodator)はフォーマットを受け取りベリファイヤは自身のフォーマットを集めたフォーマットに追加します。　それらが持っている Mizar の文法とフォーマットをリストアップしましょう。

述語（Predicates）は（原子）式（atomic formulae）の構文です、すなわち（空でもよい）項*（term）のリストに適用され、項（term）を生成します。

（*訳注 : item ->事項、term->項（数学の）と訳してあります。ご注意ください。）

述語のフォーマット（Predicate format）

〈*述語シンボル*、左引数の数、右引数の数〉

例

x ≤ y	format〈 ≤, 1, 1 〉
x,y are_isomorphic	format〈 **are_isomorphic**, 2, 0 〉
B x,y,z	format〈 **B**, 0, 3 〉

モードは（原子）型（atomic types）の構文です、すなわち（空でもよい）項（term）のリストに適用され、型（type）を生成します。

モードのフォーマット（Mode formats）

〈*モードシンボル*、引数の数〉〈*構造体シンボル*、引数の数〉

例

set	format〈 **set**, 0 〉
Subset of A	format〈 **Subset**, 1 〉
VectSpStr over A	format〈 **VectSpStr**, 1 〉

ファンクタは（原子）項（atomic terms）の構文です、すなわち（空でもよい）項（term）のリストに適用され、項（term）を生成します。

ファンクタのフォーマット（Functor formats）：

〈ファンクタ *シンボル*、左引数の数、右引数の数〉

〈*ファンクタの左カッコ述語シンボル*、引数の数、*ファンクタの右カッコ*〉

〈*選択子（selector）シンボル*、1 〉

〈*構造体シンボル*、1 〉

〈*構造体シンボル*、引数の数〉

プロセッサ（言語処理部）は最後の２つのケースを区別します、なぜなら無頓着の（forgetful）ファンクタのフォーマットと集積型（aggregating）ファンクタのフォーマットを分離して受け取るからです。

例

NAT	format〈 **NAT**, 0, 0 〉

- y	format〈 **-**, 0, 1 〉
x - y	format〈 **-**, 1, 1 〉
[: A, B :]	format〈 **[:**, 2, **:]** 〉
the carrier of A	format〈 **carrier**, 1 〉
the VectSpStr of A	format〈 **VectSpStr**, 1 〉
VectSpStr≪a,b,c,d,e≫	format〈 **VectSpStr**, 5 〉

フォーマットを導入するおもな理由はエラーのより良い診断のためです。　もしベリファイヤが構文を識別できなかったら、それは引数のフォーマットか型のどちらかが間違っていることを意味します。　Mizar はシンボルとフォーマットのオーバーライドを許します。　ときにはなぜエラーがリポートされるのか（訳注：理由を）発見するのが困難なことがあります。　２つの分離したメッセージが役にたちます。　場所を特定すること（by localization）によって環境を小さくするのに使用されるメカニズムの集合のことを言っています。　フォーマットはそのメカニズムのために使用されます。　例えば、もしシンボルが語彙目録になかったなら、**signature** 指令にリストアップされたファイルに存在していても、フォーマットは受け取られないでしょう。　Czesław Byliński は場所を絞る問題を研究しています。　近い将来この問題は Mizar 発展の主要課題になるように見受けられます。

以下にのべるのは現実の問題です。　ライブラリの開発に関して環境（Environment）はますます大きくなりつつあります。　われわれは将来環境のサイズを安定させることを望んでいます。　私が思いますに、数学の論文の知的な複雑さ（intellectual complexity）には上限があります。　私が本当に先進的な概念を使うときでさえも、複雑さは見掛け上のものでした。　例えば（let us say）私がホモロジー群を使うとき、通例は（as a rule）定義を使うことはなく、ただそれらの性質（property）のいくつかを使うだけなのです。　平均して MML の article は他の article の定理を大体 300 回参照します。　より最近の article は 30 ぐらいまでの多くの定理ファイル（Theorem Files）を使用する傾向があります。　もし article のサイズが一定なら、使用される定理ファイルの数が 1 桁大きいだけであることを意味します。　他方、場所の決定をより良くおこなうのには環境の定理ファイルが十倍小さいことが望まれます。　それゆえ、少なくともこのファイルについては、われわれは安定の近くにいるようにおもわれます。

同義語（synonyms）が許されているのでパターンと構文は区別しなければなりません。　パターンは引数の型についての情報を持つフォーマットです。　もちろんすべてのパターンが構文に適合するわけではありません。　しかし、ある構文は異なるパターンで表せるでしょうし、基本的には同じパターンが同じ

構文に使われるでしょう。　場所を限定するのに使われる他のメカニズムはフォーマットが与えられていないパターンを環境から除去するというものです。主としてシンボルの欠落のゆえに。　結局、Mizar ではシンボル、フォーマット、パターンそして構文を区別しなければなりません。

4　ある統語法パズル（A syntactic puzzle）

統語解析（syntactic analysis）における典型的な問題は異なる引数の数（arity）のファンクタとどのように戦う（cope）かです。　通常われわれはシンボルの優先度（結合の強さ）を利用してカッコの除去ができるようにします、そして同じ優先度の場合は左へグルーピングします（APL（訳注 A Programming Language）は例外で右へのグルーピングが使われます）。

これは -x・y のような例では O.K.です。　しかし x・-y のような場合にはどうすればよいのでしょう？　　基本的には２つの解決法があります：何としても優先度を使用するのには、(x・)-y を得ます、そして多分 syntactic error も得るでしょう（・シンボルとポストフィックス記法を持つ単一の項からなるファンクタでないならば）、もう一つはこの (x・)-y の解釈を可能にする特別の規則を持つことです。

問題をより良く検討するため、ある article の中でファンクタシンボル ∧ が２つのフォーマットで導入されたと考えてみましょう：　２項の演算子に対するインフィックス記法と引数のない（nullary）シンボルと、それにたいする中性要素（neutral element）（束論は良い例になります、ただし[5]で実際に使われた記法は該当しませんが）。　そういう記法では、x∧y そして ∧∧x を解釈することは容易です、しかし左方へのグルーピングは x∧∧ すなわち (x∧)∧ の場合悪い結果となります。　項 x∧∧∧y は、少なくとも人間の読者には (x∧(∧))∧y と解釈できます、しかし x∧∧∧∧y は正しくありません。　理由はとても簡単です、どの∧も偶数の項を引数としてとります（2 あるいは 0）そして１つの、すなわち奇数の項を産生します。　われわれは偶数の項から出発しました（x と y）そして正確に１つを得ることを望みます。　しかしどの ∧ もパリティーを変化させるので、奇数の ∧ が必要です。

一般的なアルゴリズムを記述するため長い項

$$(\tau_{0,1},\ldots,\ \tau_{0,k_0})\ \otimes_1\ \ldots\ \otimes_q\ (\tau_{q,1},\ldots,\ \tau_{q,k_q})$$

ここで $\tau_{i,j}$ は項で $\otimes_i$ -ファンクタ シンボルです。　あるいはより良い

$$\lambda_0\ \otimes_1\ \lambda_1\ \ldots\ \otimes_q\ \lambda_q$$

ここで λ_i は i 番目の引数のリストです。　i 番目のファンクタ シンボルに対し

ては 1<i<q に対して次のフォーマットを考えなくてはなりません。

$$(\otimes_i,\ k_{i-1},\ k_i)$$
$$(\otimes_i,\ k_{i-1},\ 1)$$
$$(\otimes_i,\ 1,\ k_i)$$
$$(\otimes_i,\ 1,\ 1)$$

最初と最後のファンクタシンボルに対しては、もちろん高々２つのフォーマットを得ます。　それは λ_{i-1}(それぞれの λ_i に関して(resp. λ_i)) が $\otimes_i$ あるいは $\otimes_{i-1}$(それぞれの$\otimes_i$ に関して(resp. $\otimes_i$))の引数であるかどうかに依存します。　これをグラフィカルに提示するために、引数のオーナーを指示する右あるいは左むきの矢印を選択しなければなりません。　それはこのようになります。

$$\lambda_0\ \otimes_1\ \lambda_1\ \ldots\ \otimes_q\ \lambda_q$$
$$\rightarrow\quad\leftarrow\qquad\leftarrow$$

矢印の向きの選択は対応する公式、すなわち

$\langle\otimes_i,\ k_{i-1},\ k_i\rangle$ を　$\lambda_{i-1}\ \otimes_1\ \lambda_i$ に対して
$\rightarrow\quad\leftarrow$

$\langle\otimes_i,\ k_{i-1},\ 1\rangle$ を　$\lambda_{i-1}\ \otimes_1\ \lambda_i$ に対して
$\rightarrow\quad\leftarrow$

$\langle\otimes_i,\ 1,\ k_i\rangle$ を　$\lambda_{i-1}\ \otimes_1\ \lambda_i$ に対して
$\rightarrow\quad\leftarrow$

$\langle\otimes_i,\ 1,\ 1\rangle$ を　$\lambda_{i-1}\ \otimes_1\ \lambda_i$ に対して
$\rightarrow\quad\leftarrow$

が許されるに違いありません。

われわれは矢印の向きから始めて連続的なフォーマットが可能かどうかチェックします。　もし $\otimes_I$ に対するフォーマットが可能でないならば、もどって $\otimes_i$ 、$\otimes_{i-1}$、... に対する右矢印の向きを変化させます。　それから左引数についてのもともとの向きと右引数のリストの新しい向きが許されるかどうか検討します。　で、もしそうならば再度先へ進むことを試みます。 もし違うなら文法エラー（syntactic error）が出るでしょう。 重要なことは、われわれが向きの変化した引数のリストを記憶しておくことです。　次回のバックトラッキングでは、われわれはこのことをこういった場所だけについて行います。　向きを２回変更するのは無意味です。　それゆえバックトラッキングを持つにもかかわらず、そのアルゴリズムはリニアです。　もしそのバックトラッキングが新しい許されたフォーマットという結果にならない場合、文法エラーとなりま

す。　不幸にもエラーの場所を正しく修正するのは不可能です。　多分それは引数のリスト向きが最後に変更された場所と（もしもあれば、なければ引数の２番目のリスト）、それから最後のバックトラッキングを引き起こした、許されていないフォーマットに対する *ファンクタシンボル*の間のどこかです。　したがって２つのエラーが報告されます： ひとつは右引数リストの向きの変更をおこした前回のバックトラッキングに対する *ファンクタシンボル*の場所に位置し、２番目のは最後のバックトラッキングを起こしたシンボルの場所に位置します。

∧ が２個の場合にどのように働くかみてみましょう。

```
x  ∧   ∧   ∧   ∧   ∧  y
→   ←   ←   ←   ←   ←
```

左へのグルーピングのため、最初のものを除くすべての矢印は右を向いています。　フォーマット（∧, 1, 0）は許されません、それゆえ２番目の矢印の向きは変更されなければなりません。

```
x  ∧   ∧   ∧   ∧   ∧  y
→   →   ←   ←   ←   ←
```

同様に４番目の矢印もまた変更されねばなりません。

```
x  ∧   ∧   ∧   ∧   ∧  y
→   →   ←   →   ←   ←
```

そして、それですべてです。　この場合、バックトラッキングはかなり短いのです。　正しい矢印の向きを与え、いくつかのカッコを挿入します。

　　x ∧ (∧) ∧ (∧) ∧ *y*

そして残りのカッコを挿入するために左へのグルーピングを使います。

　　((*x* ∧ (∧)) ∧ (∧)) ∧ *y*

∧ の数が偶数の場合、

```
x  ∧   ∧   ∧   ∧  y
→   ←   ←   ←   ←
```

そして２番目と４番目の向きを変えます。

```
x  ∧   ∧   ∧   ∧  y
→   →   ←   →   ←
```

最後に許されていない ∧ フォーマット（∧, 0, 1）を得ます。 最後のリストの向きも（最後のリストはいつも右向きです）、ひとつ前のものも（もうすでに反転されているので）変更できません、すなわちエラーになります。

もし、その結果が許されたフォーマットになる向きであるならば、どれが選ばれるかはもともとの向きに依存してたくさんあります。 矢印が低い優先度のシンボルから高い優先度のシンボルに向いているものから始めるべきでしょう、そして優先度が等しければ左に向かっているものから。 向きがもとのままならシンボルOのシークエンスで書き、逆転してあるのならばシンボルRで書くことにしましょう。 それから、その記述が辞書編集上で一番さきにくるものが選ばれた向きです。 つぎのステップで（向きは q-1 個のカッコの挿入だけを許可します）優先度が再度使用されます。 でもこんどは引数のリストの長さが 1 だけのものを取り扱います。

これはとてもスムース働きます。 私が驚くのは比較的取るに足らないようにみえる問題に対し決して些細ではない解決法をみつけることが必要なことです。

5 隠れた引数（Hidden arguments）

私は隠れた引数は LEGO で使われていると聞いたことがあるので、これは少なくとも LEGO コミュニティーにとっては新しい概念ではありません。 多分われわれが Mizar で、この隠れた引数とどのように戦ったか興味をお持ちと思います。 あるカテゴリーにおける多態性（morphism）の構成（composition）の定義について見てみましょう。 少し簡略化してあります。 その定義は non-empty Hom-sets 例えばゼロオブジェクトを持つカテゴリーだけに適合します。

```
   definition let C be Category; let a,b,c be Object of C;
   let f be Morphism of a,b;
   let g be Morphism of b,c;
(*)func g・f -> Morphism of a,c means
   ...;
  ...;
end;
```

definiens と正当性の証明（proof of correctness）は除去（omit）しています。定義ブロックの境界の内側の変数 **C,a,b,c,f,g** はローカルな定数として取り扱われます（固定された変数）。 定義ブロックの内側境界をこえた変数は実引数が代入されるはずの空いた場所を示す特別な種類の変数に変更されます

（なぜわたしが単純に自由変数と言わないのか、いぶかしんでおられるかもしれませんが、正確ではありませんが、それらはそうではありません）。　それを loci （訳注 : locus の複数形。‘座’）と呼びます。

隠れた引数を再構成するのにはパターンマッチングが使われます。　それが可能であるのを確認してください。　Mizar は隠れた loci がアクセス可能かチェックします。　定義は全く明瞭で

１．可視の loci はアクセス可能です。

２．アクセス可能な座（locus）の型に現れる座（locus）はアクセス可能です。

もちろんアクセス可能な座（loci）はこれらの２つの条件を享受（enjoy）できる座（loci）の最小の集合を構成します。

定義に従って多態性（morphism）は６個の引数をもちます。　それらの２つは可視で **g•f** のパターンが与えられていて、４つは隠されて（hidden）います。その座 **a** は、アクセス可能な座の **Object C** 型のなかに現れるのでアクセス可能です・・・といった具合です。

6　寛容性（Permissiveness）

他態性の構成（composition）の実際の定義を見てみましょう。（[2]）

```
definition let C,a,b,c,f,g;
assume A: Hom(a,b)<> ∅ & Hom(b,c)<> ∅;
func g•f -> Morphism of a,c means
:COMP:  it = g•f;
existence
proof g•f ∈ Hom(a,c) by CAT29,A; then g•f is Morphism of a,c
by CAT12;
hence thesis;
end;
uniqueness;
end;
```

loci の型は明示的に与えられていません。　これは Mizar が与えられた型の変数の識別子の予約を許すからです。　もし型が明示的に与えられていないと予約された識別子に対する型が使用されます。　多分適切な代入の後で。　この問題では重要ではなく、CAT_1 の中の予約がそうなっているので、定義の中と同じ結果が与えられます。（＊）

この定義は “idem per idem”（訳注 : 類音重畳句「私はいるといったらいるのだ！」という表現の類）エラーのように見えます。　**g•f**の構成（composition）は**g•f**

と同等と定義されています！　実際には、等号の右側の構成は以前に定義された異なる構成が使われています：

```
definition let C,f,g;
assume [g,f] ∈ dom(the Comp of C);
func g•f -> Morphism of C means
:: CAT_1:def 4
it = ( the Comp of C ), [g•f]
end;
```

私はこれを CAT_1 のアブストラクトからコピーしました。　ですから正当性のジャスティフィケーションは除外(omit)されています。　それにしても (Still) あまり面白くありません。　**g•f** はカテゴリー **C** の順序対 **[g•f]** への構成の適用の結果のように思えます。　しかし **f** と **g** の既定の型は異なります、同様のことが結果の型にもあります　― **Morphism of C** と **C** の型が異なっているとしてもです。　**Category** より（例えば構成が必ずしも結合律を満たさなくても（not necessarily associative）もっと一般的なのは **CatStr** です。　**CatStr** では Hom-set は必ずしも離接（訳注：U のこと）ではなく f ∈ Home(a,b) で g ∈ Home(b,c) であっても *g•f* が Home(a,c) に属している必要はありません。

ここでわれわれは２種類のオーバーローディングを得ます。　ひとつは *Mode Symbol* **Morphism: "Morphism of …"** と **"Morphism of …,…"** で異なったフォーマットを持つ場合です。　２番目のは構成にとって、もっと危険なものです。　実際（Actually）、わたしは今では、オーバーローディングの誤用だと思っています。　それは Czesław Byliński（CAT_1 の作者）への批判ではありません。　私自身も同様のことを NATTRA_1 でやりました。　もっとまずくやりましたが。　われわれは今のところはそれが多くのトラブルを引き起こす事を学んだだけです。

すなわち、両方の定義で仮定があるのがわかるでしょう。　最初のものでは多態性は構成(解決)可能(composable)です、２番目では Hom-set は non-empty です（一番目の後の続きのなかで明らかに早すぎる構成の定義のことを考えています、CAT_1、そしてここでは２番目として引用しています）。

最初の定義で構造体Ｃのカテゴリーのすべての多態性に対し定義された構成を得ます。　それでも、それらは構成可能でありません。　その仮定は、もしそれらが構成可能であるなら、結果は定義に従うに違いないし Composition of C をそれらに適用することを意味しています。　もしそうでなければ、結果は morphism of C です。あなたがたはこういうのが多態性であり、どれがそれだかわからないでしょう。　それがとくべつな未定義の多態性を意味しているわ

けではありません。

構成は部分関数を意味していません。　それは全関数で morphism of C の直積（Cartesian square）から morphism of C への関数と考えられます。

もし Mizar の意味論を考察（look at）するなら、この CAT_1 article でそのようなすべての関数が許されるモデルにおいて、もし構成可能な多態性のケースであるなら構成の結果と等しいと思われます。

$\mathcal{C}$を(//??be be->be) **CatStr** 型を持つすべてのオブジェクトの集合だとしましょう、そして$\mathcal{C}_0$をすべての Category の集合とします（意味論を記述するのには、もちろんタルスキー・グロタンディック集合論のモデルが存在する、より強い集合論が必要です、そして“すべてのカテゴリーの集合”は“このモデルのなかの集合を意味します、同様に意味論における他の概念も選んだモデルに相対的となります）。

$\mathcal{M}_\mathcal{C}$ を morphism of C の集合とし$\mathcal{O}_c$をCのすべてのオブジェクトの集合とします、ここでC$\in \mathcal{C}_0$です。　C$\in \mathcal{C}$、$a,b \in \mathcal{O}_c$に対して定義された morph(**C, a,b**) は Morphism of a,b 型を持つ全てのオブジェクトの集合で Hom(a,b) は通常の意味をもっています。　どこに違いがあるのでしょう？　もし Hom(a,b) が non-empty ならば、すなわち morph(C, a, b) = Hom(a,b)です。　しかし Morphism of a,b の定義もまた、permissive（寛容）です。　それは[2]に

```
definition let C,a,b;
assume Hom(a,b) <> ∅;
mode Morphism of a,b -> Morphism of C means
:: CAT_1:def 7
it ∈ Hom(a,b);
end;
```

Hom(a,b) = ∅ のときには morph(C, a, b) は恣意的な（arbitrary）空でない（non-empty）$\mathcal{M}_\mathcal{C}$ の部分集合です。　そして一番目の構成（composition）は family $\{\mathbf{comp}_C\}_{C\in\mathcal{C}}$ で、

$$\mathrm{comp}_c \ :\ \mathcal{M}_\mathcal{C} \rightarrow \mathcal{M}_\mathcal{C}$$

となるものです（such that）。

２番目の構成は family$\{\mathrm{Comp}_{c,x,y,z}\}_{C\in\mathcal{C},x,y,z\in Obc}$ で、

$\mathrm{Comp}_{C,x,y,z}$: morph(C, x, y) × morph(C,y,z) → morph(C,y,z).

それらは全く異なるものだとお分かりだと思います。　しかし、それらが意味のあるときには、それらは同じ意味を持ちます。

複雑な例からはじめました。　どのように実数の整除性（divisibility）が定義されるか見てみましょう（実際の[3]の定義とはすこし異なります）。

```
definition let x,y be Real;
assume Z: y <> 0;
func x/y -> Real means
:DIV: it · y = x;
correctness by Z;
end;
```

正当性の証明で皆さんを悩ませたくはありませんので、証明できる仮定Ｚを使うよう希望します。　その定義は、**y<>0** を知らなくても項 **x/y** 使用を許してくれます。　しかし DIV とラベルされたところに置かれた定理は

```
for x,y being Real st y <> 0
for z being Real holds z = x/y iff z·y = x;
```

それゆえ、0/0 や x/0 という項を使用する自由を持ち、さらにはそれらをの使用の理由付けをする自由をもちます。　例えば

```
x = y implies x/0 = y/0.
```

は正しいのです。　しかし遅かれ早かれ何か実質的な証明をしたくなります、DIV を参照せねばならず、そしてその前に分母が 0 に等しくないことを証明せねばなりません。
Mizar の型は non-empty の明示的意味（denotation）を持たなければなりません。　ですから “Element of …”というモードを定義するには寛容な定義（permissive definition）を使います。　それは（訳注：HIDDEN に）ビルトインされています、ですから引用することはできません。　それは以下のようになっているはずです。

```
definition
let X be set;
assume X is non-empty;
mode Element of X means
:EL: x ∈ X;
existence by AXIOMS:4;
end;
```

そういったいくらかミステリアスな **Element of Ø** という型を持つ存在(being)があります。　しかしたとえ **x is Element of Ø** だとしても **x ∈ Ø** を推論することはできません。　ちょうど反対で、Element of Ø が Ø に属さないことを、ほとんど何物も Ø に属さないということから、証明できます。　まえに、**Morphism of a,b when Hom(a,b) = Ø**　のケースで似たような状況がありました。　**EL** というラベルにおかれた、何か実質的な証明をしようとしたときに使わなければならない定理は for X being set st X is non-empty for e being set holds e is Element of X iff e∈X; を要求しています。　もちろん寛容性（permissiveness）は誤用されることがあります。　極端（extremal）な例を見るには以下の定義

```
definition
assume A: contradiction;
func kkk -> Nat means not contradiction;
existence by A;
uniqueness by A;
end;
```

を見ましょう。　それにもかかわらず正当性条件の形式（form）に関してはそれらのジャスティフィケーションは正しいのです。　それらは、もちろん contradiction の結果です。 我々はただ恣意的な記述を導入し自然数を設定(fix)します。　これは正しいのです。　しかし MML に投稿してもまずアクセプトされることはないでしょう。

7　属性（Attributes）

型の階層は木の形に似ています。　set を除いたいろいろなモードで母型は定義で与えられます。　これはある不愉快な結果を引き起こします。

A を TopSpace(topological space)として予約された識別子とします。　もし A の Open-Closed-Subset モードを定義したいのならば、それの母型がどれでありそうかを決定しなければなりません。　もし Open-Subset of A と Closed-Subset of A がすでに定義されているなら、Open-Closed-Subset of A をその両方に拡大可能にしたいと思うでしょう。　しかしそれは Mizar では不可能です。　その問題の、一つの可能な解決法は、形容詞に類似のもので型の修飾（modification）を許すようにすることです。

open-closed Subset of A

と書いて型 open、closed あるいはその両方を取り除いて拡張したいのです。

これに相当する統語（文法の）オブジェクトは Mizar では attribute とよばれます。 attribute **empty** とその反義語 **non-empty** はビルトイン(の attribute)です。 これは２つの文脈で使用可能です。 Type Expression の前の形容詞（adjective）として、例えば

non-empty set

あるいは述語（predicate）の一部として、例えば

A is non-empty。

属性（Attribute）は否定型をつくることもでき、non がそのための語として使われます。 こうして属性 empty は empty と non empty の２つの値を持ちます。 non-empty（１語彙）と non empty（２語彙）の間の違いを見てください。 この差異は non finite と infinite の違いと同じです。 いずれにしても意味は同一です。 これらの２つの可能な使用法は属性の２つの異なったスタイルの定義という結果になります：

```
definition
mode empty -> set means not ex x st x ∈ it;
existence
@proof
end;
synonym really-empty;
end;
```

あるいは

```
definition let X be set;
pred X is empty means not ex x st x ∈ X_
synonym X is_empty;
end;
```

もし最初の方法を選べば、同義語は属性であって、２番目のほうでは同義語は述語です。

すべて Mizar の型は non empty の意味（denotation）をもたなければなりませんので、以下のような型は禁止されます。

```
empty non-empty set
```

型表現の前の属性のクラスターはブール値を持つ関数で表されるので、

```
empty infinite set
```

は文法エラーとなります。　しかし

empty infinite set についてはどうしたら良いでしょう？　解決法はこの型のオブジェクトが１つでも存在すれば、全てのクラスターの証明をすることです。　それは形式上の定義です。（//??In is(formally) a definition: -> It is(formally) a definition:）

```
definition
cluster empty infinite set;
existence
proof
:: Want to use it, try to find it !
thus ex X being set st X is empty infinite;
:: Double colon in used in Mizar to start a comment.
end;
end;
```

我々はこのようなクラスターを*存在のクラスター（Existential Cluster）*と呼びます。　２番目の種類のクラスターは*条件クラスター（conditional Cluster）*でオブジェクトがある属性を持ちます（クラスターの結果のなかで与えられます）、もしそれ以外を持つと（クラスターの前項で与えられます）、例えば

```
definition
cluster empty -> finite set;
coherence @proof end;
end;
```

前項はemptyかもしれません。

```
definition let X be empty set;
cluster -> empty Subset of X;
end;
```

それは empty set の subset はまた empty であると言っています。　empty set は **Ø** で表現されます、そしてそれは空です、すなわち（//??I.e. -> i.e.）

```
Ø is empty
```

は明らかです。　あるempty setはそのempty setと等しいのですが、

```
x is empty implies X = Ø
```

は（依然として）明らかではありません（Mizar Version 3.39）。　近い将来これがビルトインされるのを期待します。　クラスターの存在を直接（immediately）証明する必要性なしにすませるために、３番目の種類の属性の定義が導入されました。

```
definition
attr empty -> set means not ex x st x ∈ it;
synonym really-empty;
end;
```

それで、存在条件は除外されています。

8　構造体と選択子（Structures and Selectors）

構造体という語で、構造体型の型を持つオブジェクトを意味します。　構造体型という語で（通常空の）項のリストへの構造体モードの適用により構築された（constructed）ある型を意味します。　構造体モードという語で*構造体定義（Structure Definition）*に導入されたあるモードを意味します。

我々が構造体を導入するとき、それはいくつかの構文を導入することを意味します。

例を見てみましょう（[4]）

```
definition let F be 1_sorted_
struct(GroupStr) VectSpStr over F ≪
carrier -> non-empty set;
add -> (BinOp of the carrier),
compl -> (UnOp of the carrier),
Zero -> Element of the carrier,
lmult-> Function of [:the carrier of F,the carrier:], the
carrier ≫;
end;
```

以下の構文が導入されています。

１．構造体モード

VectSpStr over F.

F は任意のオブジェクトで、その型は **1-sorted** に拡大され代入されるでしょう。 **Field** 型と **Ring** 型の両方とも **1-sorted** に拡張されます。 そ

れゆえ **VectSpStr over...**は線形空間の構造体で、それと同様の左要素（left modules）です。

2． 集積型ファンクタ（aggregating functor）
VectSpStr≪A, b, u, z, m≫

3． 選択子ファンクタ（A selector functor）
the limit of A, ここで A は **VectSpStr** （over something）
誰かが、なぜたった１つだけのファンクタが導入されるのか聞くかも知れません。 理由は **VectSpStr** の残りのフィールドは継承されるからです。選択子のうちのどれが新しいものかを知るために（同じ article にある）**GroupStr** の定義を見ないといけません

```
struct(LoopStr) GroupStr ≪ carrier -> nonempty set,
add -> (BinOp of the carrier),
compl ->(UnOp of the carrier),
Zero -> Element of the carrier ≫ ;
```

GroupStr は（唯一の）**VectSpStr** の先祖型としてリストにあるので、**VectSpStr** モードは **GroupStr** モードのすべての選択子を継承します。新しい構造体でたった１つ（あるいは２つ）の新しいフィールドが導入されるのは、かなり典型的なことです。 議論した例のなかで、HIDDEN 公理ファイルで導入される、たった１つのフィールド **the carrier** を持つ **1-sorted** 構造体モードについて始めましょう（このファイルは HIDDEN と呼ばれます、なぜならこのファイルはすべての article の環境にデフォルトで挿入され、その名称が指令（ファイル）にはリストアップされていないからです）。

```
definition struct 1-sorted ≪ carrier -> non-empty set ≫ ;
end;
```

ZeroStr モード、これも HIDDEN 公理ファイルで導入されますが、は先祖型と新しい選択子 **the Zero** と同様 **1-sorted** 型を持ちます。

```
definition
struct (1-sorted) ZeroStr ≪ carrier -> nonempty set,
Zero -> Element of the carrier ≫ ;
```

ZeroStr 構造体型は LoopStr モードに対する先祖型です、それは article **RLVECT_1**：

```
struct(ZeroStr) LoopStr ≪ carrier -> nonempty set,
add -> (BinOp of the carrier),
Zero -> (Element of the carrier)≫ ;
```

で導入されます、その中で新しい選択子ファンクタで２項演算子の **add** が導入されています。

4．A forgetful functor（無頓着のファンクタ）

the VectStr of A, ここで A は VecSpStr(over something)に拡張されます.

5．厳密な属性（A **strict** attribute）

Bを**1-sorted**構造体としましょう。 そうすると以下の条件は同等となります

```
A is strict VectSpStr over B
the VectSpStr of A = A
```

意味論の詳細な検討をしましょう。

選択子（selector）の概念から始めるのが一番良さそうです、あるいは全ての選択子の概念から始めるのがより良いようにも思えます。 それは原始的な概念です。 全ての選択子の集合を **Σ** で表しましょう。 もしある article でそれの有限な部分が必要であったとしても **Σ** は無限集合であると見なします。 ある人はそれが可算であり、あるいは自然数集合であるとみなすかも知れません。 もう一度いいますが、これは寛容なアプローチです、実際のところ選択子の実際の性質は重要ではありません。 重要な事実は **Σ** がしばしばリソースとよばれるものであるということです。 それは、もし article の中で選択子が導入されたなら、それは新しい選択子で、システムがそれ以前に導入されたすべての選択子とは異なるものであるのを知っていることを意味します。 しばらくの間、選択子を自然数で識別しましょう。 Mizar システムは実際にそうしています。 しかしこの識別は局所的です。 私が言っているのは、他の article では他の自然数が同じ選択子に割り振られているということです。 このように（This way）多分、より重要なことなのですが、われわれは再度寛容性を得ます。 自然数による識別が許されます、しかし証明が識別子の選択に依存してはいけません。 もちろん選択子は選択子のシンボルで識別されてはなりません。 Mizar は選択子のオーバーローディングを許します。 しかし

ライブラリ委員会は選択子のオーバーローディングを除去するという一貫したポリシーを保持しています、そこで MML ではまずそれはありません。

選択子は独立して導入されることはなく、構造体の中で導入されます、そして選択子が導入されるとき型がそれに割り当てられます。　構造体のモードが選択子の（新しい、あるいは継承した）集合（有限）と選択子への型の割り付け（assignment）を決定します。　これらの型において ***the** Selector Symbols* という項は（引数なしで）：この構造体オブジェクトのなかの選択子に対応するフィールドである、という意味を持つのを観察してください。　そういう項は構造体の定義の中だけで許されます。（**VectSpStr** のなかの）`lmult` の定義の中のある例：“台集合 F の直積（Cartesian product）からの関数”型、および台集合の台集合があります。　それは、もし F がフィールドで、我々が F 上の線形空間を扱うとき、`lmult` の変域（domain）はスカラーの直積（フィールドの台集合）とベクター（線形空間の台集合それ自身）です。

さて、ここで構造体モード μ の意味（denotation）を定義しましょう。
$\Sigma_0 = \{\sigma_1, \ldots, \sigma_k\}$ は μ に 関する選択子の集合としましょう、そして

$$\Theta_0 : \Sigma_0 \rightarrow \Theta$$

は型に割り当てましょう。　μ の意味（denotation）はすべての部分関数から構成され

$$f : \Sigma \rightarrow \Omega$$

は２つの条件が約束（enjoy）されています

1. $\Sigma_0 \subseteq \mathrm{dom}(f)$
2. 選択子に割り当てられたオブジェクトは適切である（ここでは技術的な理由で定義できません；その意味が明瞭であることを望みます）

(**)　　$\Sigma_0 = \mathrm{dom}(f)$

であることを要求していないことを注意しましょう。
以前のバージョンの Mizar ではその要求がありました。　それは明かに問題でした。　**VectSpStr** は、もし(**)が満たされているなら、**GroupStr** まで拡張されません。　なぜなら **GroupStr** の意味は４要素の変域から構成され、**VectSpStr** の意味は５要素の変域の関数だからです。　現在使用されている Mizar の意味論ではこの拡張は許されています。

構造体 μ に関する属性 **strict** は、要求(**)が満たされていることを意味します。

無頓着のファンクタ the μ of A は、もし A が関数 f を表すなら、$f \mid \Sigma_0$ の制限を表します。　それゆえ結果はいつも strict です。

選択子 σ_i に割り当てられた集積ファンクタ（aggregating functor）$\mu \ll A_1, \cdots, A_k \gg$ は、A_i であらわされるオブジェクトは明示的に関数 f を与えます。それもまた strict です（申し訳ありません、これは正確ではありません。　構造体のモードは選択子の集合だけを決定します。　集積型構造体に対しては、選択子のリストが関係あります、そして集積のなかの引数の順番は構造体の定義のフィールドの順番と同一でなければなりません。　それゆえ、継承された選択子は構造体定義のなかで繰り返す必要があります）。

References

[1] Grzegorz Bancerek. The fundamental properties of natural numbers. *Formalized Mathematics,* 1(**1**):41-46, 1990

[2] Czesław Byliński. Introduction to categories and functors. *Formalized Mathematics*, 1(**2**):409-420, 1990.

[3] Krzysztof Hryniewiecki. Basic properties of real numbers. *Formalized Mathematics,* 1(**1**):35-40, 1990.

[4] Eugeniusz Kusak, Wojciech Leończuk and Michał Muzalewski. Abelian groups, fields and vector spaces. *Formalized Mathematics,* 1(**2**):335-342, 1990.

[5] Stanisław Żukowski. Introducton to lattice theory. *Formalized Mathematics,* 1(**1**):215-222, 1990.

興味があったので訳出してみましたが、力及ばず、内容が理解できない個所が多すぎてgive upしました。　廃棄するのも悔しいので公開します。　あまり改訂の意欲もないので役に立つ方、内容の理解できる方改変して御使用ください。マニュアルではなくscientific articleでreviewの形式なのでcopy rightはdon't careです。

2010 早春　　l'Hospitalier