Some Features of the Mizar Language

Andrzej Trybulec
Institute of Mathematics
Warsaw University in Białystok
15-267 Białystok, Akademicka 2, Poland

Preface. It is impossible, in a short article, to present completely the Mizar language. I have chosen some topics that seem to be interesting or give some insight into our work. I devoted a large part of this presentation to the technical problems related to the lexical analysis (tokenization), identification of constructors and similar topics. They are quite often unfairly neglected. Correct solution of such problems have a big impact on the practical value of the system. Also, I hope that we may standardize solutions for such problems. I would start with a law stating what characters must be used to start a comment (and with an International Agency able to enforce this law).

I am so accustomed with various abbreviations used in the Mizar community, that I am afraid that I use them without any explanation. SUM (Polish: Stowarzyszenie Użytkowników Mizara, literally: the Association of Mizar Users) is a small scientific society (about 40 members, 4 circles, three in Poland, one in Japan) based in Poland and called usually in English – the Mizar Society. The main activity of SUM is the collection of Mizar articles. The Library Committee of SUM is in charge of accepting and revising the articles. The accepted articles constitute MML – the Main Mizar Library. MizUG – Mizar Users Group is an organization based in Belgium. It is in charge of distributing the Mizar System and publishing Formalized Mathematics.

I adopted some rules that make easier reading the text. Fragments of Mizar articles are in the `tt` font. Including separate Mizar symbol, e.g. `set`. The italic is used for names of Mizar grammatical constructions. If I have used them in a skeletal Mizar text I rather use boldface for reserved words of Mizar and roman otherwise.

Contents

1 Introduction

A Mizar *Article* consists of two parts : the *Environment Declaration* and the *Text Proper*

The *Environment Declaration* begins with **environ** and consists of *Directives*. The *Text Proper* is a finite sequence of *Sections*, every *Section* begins with **begin** and consists of *Items*. The division of the *Text Proper* into *Sections* is only for editing purposes and has no impact on the correctness of the article.

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

The two parts of the *Article* are processed by two different programs: Accomodator and Verifier. Accomodator processes the *Environment Declaration* and creates the Environment consisting of a number of files in which the knowledge requested in *Directives* and imported from the Data Base is stored. Verifier has no direct communication with the Data Base and verifies the correctness of the *Text Proper* using only knowledge stored in the Environment.

There are two kinds of *Directives*: *Vocabulary Directives* and *Library Directives*.

Vocabulary Directives request symbols from vocabularies. A vocabulary is an ASCII file on which symbols are defined. The *Vocabulary Directives* has the form

vocabulary *Vocabulary Name* , . . . , *Vocabulary Name* ;

Library Directives request from the Data Base particular information. Unfortunately just now we started to change them. The directives that request the conceptual framework used in an article is written now

signature *Article Name* , . . . , *Article Name* ;

in the future will be separated into two

notations *Article Name* , . . . , *Article Name* ;

constructors *Article Name* , . . . , *Article Name* ;

The reason is that it is allowed in Mizar to introduce synonyms (and antonyms if they are meaningful) if one does not like original notation. Therefore the constructor (predicate, functor and so on) may be represented in different ways. The **constructors** directive will inform the Accomodator which articles should be used to create the conceptual framework for our article, the **notations** directive will inform from which articles the notation should be borrowed.

The remaining directives are :

clusters *Article Name* , . . . , *Article Name* ;

(the meaning of this will be explained later) and

definitions *Article Name* , . . . , *Article Name* ;

that request definientia, if one wants to use them in proving by definitional expansion. It is allowed in Mizar to use proofs of the form

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

To be able to verify such proof the Verifier must know the definiens of the inclusion.

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

It enables referring to the theorems in the listed articles. Such references are used in *Straightforward Justifications*, that has the form

by*Reference*, . . . , *Reference*

Reference is either a *Private Reference*, a reference to a statement in the article or a *Library Reference*, a reference to a theorem stored in the Data Base. It has two forms: *Article Name* : *Theorem Number* or *Article Name* : **def** *Definition Number*. The second one refers to the so called definitional theorem, a theorem automatically created that describes the meaning of the definition. It is a source of some confusion. To get access to definitional theorems in an article one has to insert its name into the **theorems** directive rather than the **definitions** directive.

schemes *Article Name* , . . . , *Article Name* ;

It is used to get access to schemes, which are theorems in which second order free variables occur. The good example is the scheme of induction [1]. To justify by this scheme a theorem of the form "for every natural number the property α holds" one must first prove

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

and then it is possible to use the scheme :

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

The next program to be considered is the Extractor. It extracts from the *Article* public information (private information such as justifications or lemmas is ignored) and stores it in library files. Those files are used by the Accomodator to create an Environment for the next article.

Data Base consists of two parts: Public Data Base and Private Data Base. Public Data Base distributed by the Mizar Society consists of library files created using the articles submitted to Main Mizar Library (MML), Private Data Base may be created by a user (or a group of users) using Extractor on articles stored in Private Library. However, the library files when transferred to Public Data Base, are optimized. Using a big Private Library may cause some troubles and it is a good policy to submit an article to MML as soon as the author is certain of its quality (for that is usually necessary to write a couple of articles using it).

I do not want to be too detailed in the description of the *Text Proper*. I restrict myself to the description of only those constructions whose names are used in the sequel.

Blocks consists of *Items*. They have an introducing *Reserved Word*, different for different kinds of *Blocks*. Some of them end with **end**. the *Proof* has an **end**, the *Section* has not. A *Definitional Block* is a *Block* that is used to introduce definitions. Its syntax:

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

Items can be easily recognized because they always end with a semicolon. Moreover every semicolon ends an *Item*. In different *Blocks* different sets of *Items* are used. In a *Section*, *Text Items* are allowed, in a *Definition* – *Definitional Items*. A *Definition* consists of a *Definitional Block* juxtaposed with a semicolon. It is a *Text Item*, but not a *Definitional Item*, *Definitional Blocks* cannot be nested. A *Structure Definition* is both a *Text Item* and a *Definitional Item*. Its syntax:

struct (*Ancestors*)
Structure Symbol **over** *Loci*
$\ll$ *Field Segment*, . . . ,*Field Segment* $\gg$

Two parts: (*Ancestors*) and **over** *Loci* are optional and may be omitted. Actually, if a *Structure Definition* is used as a *Text Item*, then **over** *Loci* must be omitted. A *Field Segment* is

Selector Symbol, ..., *Selector Symbol* −> *Type Expression*
A selector that occurs in a structure is called its field.

2 Lexicon

Lexicons of different articles may be different. The lexicon depends on the lexical context, i.e. vocabularies requested in the *Vocabulary Directives*. It consists of the following sets of tokens:

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
```

We distinguish *Lexems* and tokens. There are in the vocabulary synonymous tokens:

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

(the last two only in some contexts) that represent the same *Lexem*, and homonymous tokens, that in different contexts represent different *Lexems*. They are

`func pred set = { } [ ]`

with the exception of the first two, they may be used as symbols. Some tokens are reserved for future use and now are not used at all. They are

`define selector aggregate prefix`

Some *Lexems* cannot be used in an article submitted to MML. They are

`canceled @proof`

`canceled` is used to replace a theorem that is removed from an article in MML, to keep correct numeration of theorems, and `@proof` suspends the verification of the proof (this mechanism, proposed by Zbigniew Karno is used only to speed up the verification of an article).

Property Names

```
symmetry reflexivity irreflexivity
associativity transitivity commutativity
```

Only first three are supported as yet. And the implementation could be better.

Symbols

As it was explained before they are given on vocabularies. Only those are included in the Lexicon of an article that are either on vocabularies which names are listed in *Vocabulary Directive* or on the HIDDEN vocabulary or are homonyms. Every line in the vocabulary begins with one character qualifier that determines the kind of the symbol (*Predicate Symbol*, *Functor Symbol* and so on) followed immediately by the representation of the symbol. The following qualifiers are used :

R	for *Predicate Symbol*
O	for *Functor Symbol*
M	for *Mode Symbol*
G	for *Structure Symbol*
U	for *Selector Symbol*
V	for *Attribute Symbol*
K	for *Left Functor Bracket*
L	for *Right Functor Bracket*

For a *Functor Symbol* an additional information is given on vocabulary: the priority that is a number between 0 and 255 (64 by default).

Homonymous symbols (homonyms for reserved words) are recognized by the Verifier and are on no vocabulary. They are the following:

`set`	this is a *Mode Symbol*
`=`	this is a *Predicate Symbol*
{ }	these are functor brackets
[ ]	these are functor brackets, too

You might be surprised by the unnecessary complexity of it. It is at least partially the result of the evolution of Mizar and the different preferences of different authors.

Numerals

0 is a *Numeral* and a sequence of digits starting with a non zero digit is a *Numeral*

Article Names

Only *Article Names* given in the **theorems** directive are inserted in the lexicon of an article.

Scheme Names

They have the form of *Identifiers*. The fact that they are not private is somewhat irregular. It is necessary to do something with it in the future. Czesław Byliński supports the idea that they must be referenced by a sort of *Library Reference*. Only those *Scheme Names* are accessible that where introduced in articles which names are listed in the **schemes** directive.

Identifiers

Any sequence of letters (Mizar distinguishes case), digits, apostrophe and underscore sign that does not belong to any previous class of symbols. Please note that the concept of the *Identifier* depends on the Environment used. The policy of the Library Committee is that symbols that look like *Identifiers* must have at least three characters. Unfortunately the *Functor Symbol* `U` was introduced for the union of two sets (therefore in most articles it cannot be used as an *Identifier*), `Ü` (introduced in the vocabulary HIDDEN) is not used at all but it is supposed to replace `U` in the future. Before implementing the above mentioned policy geometry folk succeeded to introduce one letter *Predicate Symbols* for betweenness and collinearity, we plan to press them to change those symbols.

Invisible characters (LF CR SP) cannot be used inside tokens. However, it is not necessary to separate tokens by them. If they are not separated they create a sequence of characters that must be sliced into tokens. The rule is simple, the slicer (the procedure that does it) tries to get a longest possible token and cut it off. Then the process is repeated without backtracking. If no prefix of such segment is a legal token then a tokenization error is reported (with the message "Unknown Token") and one character is cut off.

Such rules of slicing cause sometimes problems. E.g. for inclusion the token `c=` is used. So if one wrote
`a` $\cap$ `c U c= c`
a syntactic error would occur, because `c=` would be treated as the inclusion symbol. The cure is simple. Just insert a space
`a` $\cap$ `c U c = c`
Still, even for experienced user, it is sometimes confusing.

You might wonder why I devoted so much place for seemingly trivial matters. The problems that we have with the lexicon are, I do believe, for a practical evolving system that must be convenient and flexible. They occur also on more sophisticated levels. Only on those levels it is not so easy to explain them. I wanted to use lexical level as an illustration for the problems that we have to fight with.

3 Notation

The format describes with how many arguments a Constructor Symbol may be used. Accomodator collects formats and Verifier appends own formats to the collection. Let us list Mizar constructors and formats that they have.

Predicates are constructors of (atomic) formulae, i.e applied to a (possibly empty) list of terms, create a term.

Predicate format
⟨ *Predicate Symbol*, Left Arguments Number, Right Arguments Number ⟩

Examples

`x` $\leq$ `y`	format ⟨ $\leq$, 1, 1 ⟩
`x,y are_isomorphic`	format ⟨ `are_isomorphic`, 2, 0 ⟩
`B x,y,z`	format ⟨ `B`, 0, 3 ⟩

Modes are constructors of (atomic) types, i.e applied to a (possibly empty) list of terms, create a type.

Mode formats
⟨ *Mode Symbol*, Arguments Number ⟩ ⟨ *Structure Symbols*, Arguments Number ⟩

Examples

`set`	format ⟨ `set`, 0 ⟩
`Subset of A`	format ⟨ `Subset`, 1 ⟩
`VectSpStr over A`	format ⟨ `VectSpStr`, 1 ⟩

Functors are constructors of (atomic) terms, i.e applied to a (possibly empty) list of terms, create a term.

Functor formats:
⟨ *Functor Symbol*, left arguments number, right arguments number ⟩
⟨ *Left Functor Bracket*, arguments number, *Right Functor Bracket* ⟩
⟨ *Selector Symbol*, 1 ⟩
⟨ *Structure Symbol*, 1 ⟩
⟨ *Structure Symbol*, arguments number ⟩

The processor can distinguish two last cases because it collects separately formats for forgetful functors and for aggregating functors.

Examples

`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 ⟩

The main reason for introducing formats was to enable better diagnostics of errors. If the Verifier cannot identify a constructor it means that either the format or the types of arguments are wrong. Mizar allows for overloading both of symbols and of formats. Sometimes it is difficult to find out why an error is reported. Two separate messages help.

We mean by localization a set of mechanisms that are used to make the Environment smaller. Formats are used for such mechanisms. For instance, if a symbol is not in the lexicon, formats in which it occurs are not collected, even if they are present on files listed in the **signature** directives. Czesław Byliński works on the problem of the localization. It seems that in near future it will be the main issue in the development of Mizar.

That is the real problem. With the development of the library the Environments tend to get bigger and bigger. We hope to stabilize the size of the Environments in the future. The intellectual complexity of a mathematical paper has, I do believe, an upper limit. Even when I use really advanced concepts the complexity is apparent. When I use, let us say, the homology groups, I do not use, as a rule, the definition but only some their properties. On the average an article in the MML refers to theorems in other articles approximately 300 times. More recent articles tend to use many Theorem Files, up to 30. If the size of articles is steady, it means that the number of theorem files used will be only one order greater. On the other hand there are hopes that with better localization the theorem file in the Environment will be ten times smaller. Therefore it seems that we are close to stabilize at least this file.

Because synonyms are allowed one has to distinguish patterns and constructors. Pattern is a format with the information on types of arguments. Not every pattern fits a constructor, of course. But a constructor may be represented by different patterns, and basically the same pattern may be used for the same constructor. Another mechanisms used for the localization is to eliminate from the Environment those patterns for which no Format was given. Mainly because of the lack of the symbol.

Eventually in Mizar one has to distinguish symbol, format, pattern, and constructor.

4 A syntactic puzzle

A typical problem for syntactic analyses is how to cope with functor symbols of different arity. Usually, we allow omitting parenthesis using the priority of symbols (the force of binding) and in the case of symbols of equal priority, grouping to the left (with APL being an exception, in which grouping to the right is used).

It is O.K. in such examples as - x · y. But what should we do with x · - y? There are basically two solutions: to use priority anyway, and then you get (x ·) - y and probably a syntactic error (if there is no unary functor with the · symbol and postfix notation) or to have special rules that enable to construe this term as x · (- y)

To better see the problem let us suppose that in an article a functor symbol $\wedge$ was introduced with two formats: in infix notation for binary operation and as a nullary symbol for the neutral element of it (lattice theory could be a good example, even if actual notation used in [5] is different). In such notation it is easy to construe $x \wedge y$ and $\wedge \wedge x$, but grouping to the left gives the wrong result in the case of $x \wedge \wedge$ namely $(x \wedge)\wedge$. The term $x \wedge \wedge \wedge y$ can be construed, at least by a human reader, as $(x \wedge (\wedge)) \wedge y$ but $x \wedge \wedge \wedge \wedge y$ is incorrect. The reason is quite simple, every $\wedge$ takes an even number of terms as arguments (two or zero) and produces one, i.e. an odd number of terms. We start with an even number of terms (x and y) and want to get exactly one. But every $\wedge$ changes the parity, therefore we need an odd number of $\wedge$'s.

To describe the general algorithm let consider us a long term of the form

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

where $\tau_{i,j}$'s are terms, and $\otimes_i$ - functor symbols, or better

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

where λ_i is the i-th list of arguments. For the i-th functor symbol we have to consider the following formats

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

for $1 < i < q$. For the first and the last functor symbol we get of course at most two formats. It depends whether λ_{i-1} (resp. λ_i) are arguments of $\otimes_i$ or $\otimes_{i-1}$ (resp. $\otimes_{i+1}$). To present it graphically we have to choose arrows oriented to the right or to the left, pointing to the owner of the arguments. It looks like this

$$\begin{array}{ccccc} \underset{\longrightarrow}{\lambda_0} & \otimes_1 & \underset{\longleftarrow}{\lambda_1} & \ldots & \otimes_q & \underset{\longleftarrow}{\lambda_q} \end{array}$$

the choice of the orientation of arrows must be such that the corresponding format, i.e

$$\begin{array}{lllll} \langle\otimes_i, k_{i-1}, k_i\rangle & \text{for} & \underset{\longrightarrow}{\lambda_{i-1}} & \otimes_1 & \underset{\longleftarrow}{\lambda_i} \\ \langle\otimes_i, k_{i-1}, 1\rangle & \text{for} & \underset{\longrightarrow}{\lambda_{i-1}} & \otimes_1 & \underset{\longrightarrow}{\lambda_i} \\ \langle\otimes_i, 1, k_i\rangle & \text{for} & \underset{\longleftarrow}{\lambda_{i-1}} & \otimes_1 & \underset{\longleftarrow}{\lambda_i} \\ \langle\otimes_i, 1, 1\rangle & \text{for} & \underset{\longleftarrow}{\lambda_{i-1}} & \otimes_1 & \underset{\longrightarrow}{\lambda_i} \end{array}$$

must be allowed.

We start with an orientation of arrows and check if the consecutive formats are allowed. If the format for $\otimes_i$ is not, then we backtrack, changing the orientation of right arrows for $\otimes_i, \otimes_{i-1}, \ldots$ and then we check whether the format with the original orientation of left arguments and new orientation of the list of right arguments is allowed. If so, we try to go forward again. If not we get a syntactic error. What is important is that we must remember the list of arguments which orientation has been changed. Backtracking next time, we do it only to this place. It is meaningless to change the orientation twice. Therefore the algorithm notwithstanding with backtracking is linear. If the backtracking does not result in a new allowed format we get a syntactic error. Unfortunately it is impossible to fix the place of the error. Probably it is somewhere between the last change of the orientation of the list of arguments (if any, if none, then the second list of arguments) and the *Functor Symbol* for which we get not allowed format that caused the last backtracking. Accordingly two errors are reported: one placed at the *Functor Symbol* for which the previous backtracking changed the orientation of the right list of arguments, the second one placed at the symbol that caused the last backtracking.

Let us look how it works in the case of two $\wedge$'s.

$$\begin{array}{ccccccccccc} \underset{\longrightarrow}{x} & \wedge & \underset{\longleftarrow}{} & \wedge & \underset{\longleftarrow}{} & \wedge & \underset{\longleftarrow}{} & \wedge & \underset{\longleftarrow}{} & \wedge & \underset{\longleftarrow}{y} \end{array}$$

Because of the grouping to the left, all arrows but the first one are oriented to the right. The format ($\wedge$, 1, 0) is not allowed therefore the orientation of the second arrow must be changed :

$$\begin{array}{ccccccccccccc} x & & \wedge & & \wedge & & \wedge & & \wedge & & \wedge & & y \\ & \longrightarrow & & \longrightarrow & & \longleftarrow & & \longleftarrow & & \longleftarrow & & \longleftarrow & \end{array}$$

similarly the orientation of the fourth arrow must be changed also

$$\begin{array}{ccccccccccccc} x & & \wedge & & \wedge & & \wedge & & \wedge & & \wedge & & y \\ & \longrightarrow & & \longrightarrow & & \longleftarrow & & \longrightarrow & & \longleftarrow & & \longleftarrow & \end{array}$$

and that is all. In this case backtracking is rather short. Given the proper orientation of arrows we may insert some parenthesis

$$x \wedge (\wedge) \wedge (\wedge) \wedge y$$

and using grouping to the left to insert the remaining parenthesis

$$((x \wedge (\wedge)) \wedge (\wedge)) \wedge y$$

In the case of an even number of $\wedge$'s

$$\begin{array}{ccccccccccc} x & & \wedge & & \wedge & & \wedge & & \wedge & & y \\ & \longrightarrow & & \longleftarrow & & \longleftarrow & & \longleftarrow & & \longleftarrow & \end{array}$$

and changing the second and the fourth orientation

$$\begin{array}{ccccccccccc} x & & \wedge & & \wedge & & \wedge & & \wedge & & y \\ & \longrightarrow & & \longrightarrow & & \longleftarrow & & \longrightarrow & & \longleftarrow & \end{array}$$

we get for the last $\wedge$ format ($\wedge$, 0, 1) which is not allowed. We cannot change neither the orientation of the last list (the last list is always oriented to the right) nor the previous one (because it is already reversed), i.e. we get an error.

If there exists an orientation which results in allowed formats, that there are many which one is chosen depends of the original orientation. One must start with such in which arrows are oriented from the symbols of lower priority to the symbols of higher priority, and to the left if priorities are equal. Let us describe an orientation by a sequence of symbols O if the orientation is original (according to priorities) and R if it is reversed. Then the orientation chosen is this one which the description is earliest in lexicographical order. In the next step (the orientation allows only for inserting $q-1$ parenthesis) the priority is used again. But now we deal only with the list of arguments of the length 1.

It works quite smoothly. What seems to be surprising for me is that for comparatively trivial problems it was necessary to find a not so trivial solution.

5 Hidden arguments

I was told that hidden arguments are used in LEGO, so it is not a new concept, at least for the LEGO community. You might be interested how we cope with them in Mizar. Let us look at the definition of the composition of morphisms in a category. Slightly simplified. Such definition fits only categories with non-empty Hom-sets, e.g. with a zero object.

```
       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;
```

I have omitted the definiens and the proof of the correctness.

The variables `C,a,b,c,f,g` in the limits of the *Definitional Block* are treated as local constants (fixed variables). Beyond the limits of the *Definitional Block* that are changed to the special kind of variables representing empty holes in that actual arguments may be substituted (You may wonder why I do not say free variables simply. Well, they are not. Not exactly). We call then loci.

The pattern matching is used to reconstruct the hidden arguments. To be sure that it is possible, Mizar checks the accessibility of hidden loci. The definition is quite obvious
1. the visible loci are accessible
2. the locus that occurs in the type of an accessible locus is accessible
Of course the accessible loci constitute the smallest set of loci that enjoys these two conditions.

According to the definition the composition of morphisms has 6 arguments. Two of them are visible and given in the pattern `g·f`, four are hidden. The locus a is accessible because occurs in the type `Morphism of a,b` of an accessible locus `f`, `C` is accessible because occurs in the type `Object of C` of an accessible locus a and so on.

6 Permissiveness

Let us look at the actual definition of the composition of the morphisms ([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;
```

The types of loci are not given explicitly. This is so because Mizar allows for the reservation of *Identifiers* for variables of a given type. If the type is not given explicitly, then the type for which the *Identifier* has been reserved is used. Perhaps after proper substitutions. It does not matter for the problem in question and the reservation in CAT_1 is such that gives the same result as in the definition $(*)$

The definition looks like "idem per idem" error. The composition `g·f` is defined as equal to `g·f` ! In fact, the composition used on the right side of the equality is a different composition defined earlier:

```
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;
```

I have copied it from the abstract of CAT_1. So the justification of correctness is omitted. Still, it is not very interesting. It means that `g·f` is the result of the application of the composition in the Category `C` to the ordered pair `[g,f]`. But, the default types of `f` and `g` are different, the same as the type of the result - `Morphism of C` and even the type of `C` is different. It is `CatStr`, something that is more general than the `Category` (for instance the composition is not necessarily associative). In `CatStr` Hom-sets are not necessarily disjoint and even if $f \in \mathrm{Hom}(a,b)$ and $g \in \mathrm{Hom}(b,c)$, it is not necessary for $g{\cdot}f$ to belong to $\mathrm{Hom}(a,c)$.

We get two kinds of overloading here. One in the case of the mode with *Mode Symbol* `Morphism`: "Morphism of ..." and "Morphism of ..., ...", with different formats. And the second, more dangerous, for the composition. Actually, I think now, that it is a misuse of overloading. It is not a criticism of Czesław Byliński (the author of CAT_1). I did the similar thing in NATTRA_1. Even worse. We just learned in the meantime that it causes too much troubles.

To wit, you may observe that in both definition there are assumptions. In the first one that the morphisms are composable, in the second one that the Hom-sets are non empty. (In the sequel by first I mean the definition of the composition that is really earlier, in CAT_1, and that is cited here as second).

In the first definition we get the composition that is defined for all morphisms of a category structure C. Even if they are not composable. The assumption means that if they are composable, the result must be according to the definition, the application of the composition of C to them. If they are not, the result is a morphism of C. You know that it is a morphism, you do not know which one. It does not mean that it is a special undefined morphism.

It does not mean that the composition is a partial function. It is total, considered as a function from the Cartesian square of the morphisms of C to the morphisms of C.

If we look at the semantics of a Mizar article, in this case the CAT_1 article, it means that in the model every such function is allowed, if in the case of composable morphisms it is equal to the result of the composition.

Let $\mathcal{C}$ be be the set of all objects that have the type `CatStr` and let $\mathcal{C}_0$ be the set of all categories (to describe semantics we need of course stronger set theory in which a model of Tarski Grothendieck set theory exists, and "the set of all categories" means "the set of all categories in this model", similarly the other semantic concepts must be relative to the model chosen).

Let M_C, where $C \in \mathcal{C}_0$, be the set of all morphisms of C and O_C be the set of all objects of C. $\mathrm{morph}(C,a,b)$ defined for $C \in \mathcal{C},\ a,b \in O_C$ is the set of all object that have type `Morphism of a,b` and $\mathrm{Hom}(a,b)$ has the usual meaning. Where is the difference? If $\mathrm{Hom}(a,b)$ is non-empty, then $\mathrm{morph}(C,a,b) = \mathrm{Hom}(a,b)$. But the definition of `Morphism of a,b` is permissive, too. It is in [2]

```
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;
```

When $\mathrm{Hom}(a,b) = \emptyset$, the $\mathrm{morph}(C,a,b)$ is an arbitrary non empty subset of M_C.

Then the first composition is a family $\{\mathrm{comp}_C\}_{C\in\mathcal{C}}$, such that

$$\mathrm{comp}_C : M_C \rightarrow M_C.$$

The second composition is a family $\{\mathrm{Comp}_{C,x,y,z}\}_{C\in\mathcal{C}_0,\ x,y,z\in Ob_C}$ such that

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

One can see that they are quite different. However, when they are meaningful, they have the same meaning.

We started with a complicated example. Let us see how the divisibility of real numbers is define. (The actual definition in [3] is a bit different).

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

I do not want to bother you with the proof of correctness, I hope you agree that using the assumption `Z` one can prove it. The definition allows the use of the term `x/y` even if one does not know if `y <> 0`. However, the theorem placed at the label DIV says that

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

Therefore one has freedom to use such terms as `0/0` or `x/0` and even to reason using them. It is true for instance that

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

But sooner or later one wants to prove something substantial, then must refer to `DIV` and before that must prove that the denominator does not equal 0.

The types in Mizar must have non empty denotation. Therefore to define the mode "Element of ..." we use a permissive definition. It is built-in, so I cannot cite it. It could look like that

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

There exists such somewhat mysterious being that has the type `Element of` $\emptyset$ But even if `x is Element of` $\emptyset$ one cannot infer that `x` $\in \emptyset$. Just opposite, one can prove that element of $\emptyset$ does not belong to $\emptyset$, mostly because nothing belongs to $\emptyset$. We met a similar situation before, in the case of `Morphism of a,b` when `Hom(a,b) =` $\emptyset$.

The theorem placed at the label `EL` that must be used if one wants to prove something substantial claims that

for X being set st X is non-empty for e being set holds e is Element of X iff e $\in$ X;

Of course the permissiveness can be misused. To see an extremal example let us look at the definition

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

Notwithstanding with the form of the correctness conditions the justification of them is correct. They are consequences of the contradiction, of course. We just introduced notation for an arbitrary but fixed natural number. This is correct. But it hardly may be accepted in an article submitted to the MML.

7 Attributes

The hierarchy of types is tree like. For every mode, with the exception of the mode `set`, the mother type is given in the definition. This causes some unpleasant consequences.

Let `A` be an *Identifier* reserved for `TopSpace` (topological space). If we want to define mode `Open-Closed-Subset of A` we must decide what will be the mother type of it. If both `Open-Subset of A` and `Closed-Subset of A` have been already defined, we would like to be able to widen `Open-Closed-Subset of A` to both. But it is impossible in Mizar. One possible solution to the problem is to allow the modification of a type with something that resembles adjectives. We would like to write
`open closed Subset of A`
and widen the type by omitting `open` or `closed` or both.

The corresponding syntactic object is called in Mizar an attribute. The attribute `empty` with an antonym `non-empty` is built-in. One may use it in two contexts. Either as an adjective before a *Type Expression*, e.g.
`non-empty set`
or as a part of predicate, e.g
`A is non-empty`.
Attributes may be negated, `non` is used as the word for it. Hence the attribute `empty` has two values `empty` and `non empty`. Please observe the difference between `non-empty` (one *Lexem*) and `non empty` (two *Lexems*). The difference is the same as the difference between `non finite` and `infinite`. The meaning is the same, anyway. These two possible uses resulted in two different styles of the definition of an attribute: either

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

or

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

If we choose the first way, that synonym is an attribute, if the second the synonym is a predicate.

All types in Mizar must have non empty denotation, therefore we must prohibit such types as
`empty non-empty set`
Because the cluster of attributes before the *Type Expression* is represented as a Boolean valued function, so `empty non-empty set` is a grammatical error. But what we should do with
`empty infinite set` ?

The solution is to prove for every cluster that at least one object of this type exists. In 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;
```
We call such a cluster an *Existential Cluster*. The second kind of clusters is *Conditional Cluster* that says that an object has some attributes (given in the *Consequent* of the cluster), if it has some other (given in the *Antecedent* of the cluster). E.g.
```
definition
cluster empty -> finite set;
coherence @proof end;
end;
```

The antecedent may be empty:
```
definition let X be empty set;
cluster -> empty Subset of X;
end;
```
it says that a subset of an empty set id empty. The empty set is denoted by ∅, and it is empty. I.e.
```
∅ is empty
```
is obvious. An empty set is equal to the empty set, but
```
X is empty implies X = ∅
```
is (still) not obvious (Mizar Version 3.39). We hope to built-in it in the nearest future.

To dispense with the necessity of proving the existence of the cluster immediately the third sort of attribute definition was introduced:
```
definition
attr empty -> set means not ex x st x ∈ it;
synonym really-empty;
end;
```

Then the **existence** condition is omitted.

8 Structures and Selectors

By a structure we mean an object that has a structure type. By a structure type we mean a type constructed by application of a structure mode to a (usually empty) list of terms. By a structure mode we mean a mode introduced in a *Structure Definition.*

When we introduce a structure it means that we introduce a number of constructors. Let us look at the example ([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;
```

The following constructors are introduced :

1. A structure mode
   `VectSpStr over F.`

   For F any object, which type widens to `1-sorted` may be substituted. Both `Field` type and `Ring` type widens to `1-sorted`. Therefore `VectSpStr over` ... is the structure of linear spaces and left modules as well.

2. An aggregating functor
   `VectSpStr≪ A, b, u, z, m ≫`

3. A selector functor
   `the lmult of A`, where A must have a type that widens to `VectSpStr` (over something)

   One may ask why only one selector functor is introduced. The reason is that the remaining fields of the `VectSpStr` are inherited. To see which of the selectors are new one has to look at the definition of `GroupStr` (in the same article).

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

   Because `GroupStr` is listed as (the only) ancestor of `VectSpStr`, the `VectSpStr` mode inherits all selectors of the `GroupStr` mode. It is rather typical, that in a new structure just one (or two) new fields are introduced. In the discussed example, one has to start with the `1-sorted` structure mode that has only one field `the carrier` that is introduced in HIDDEN axiomatic file (it is called HIDDEN, because it is inserted in the environment of every article by default, and its name is not listed in the directives.

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

   The `ZeroStr` mode, introduced also in HIDDEN axiomatic file, has the `1-sorted` type as an ancestor and a new selector `the Zero`.

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

   The `ZeroStr` structure type is an ancestor for the `LoopStr` mode, that is introduced in the article RLVECT_1 :

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

   in which a new selector functor `add` that is a binary operation on the carrier has been introduced. And it is the ancestor of the `GroupStr` mode.

4. A forgetful functor
   `the VectSpStr of A`, where A must have a type that widens to `VectSpStr` (over something)

5. A `strict` attribute

Let B be a `1-sorted` structure. Then the following conditions are equivalent

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

Let us try to have a closer look at the semantics.

It seems that the best way is to start with the notion of a selector, or better with the notion of the set of all selectors. It is a primitive notion. Let us denote the set of all selectors by Σ. We assume that Σ is infinite, even if in one article we need only a finite part of it. One may assume that it is countable or even that it is just the set of all natural numbers. It is again a permissive approach, actually it does matter what is the actual nature of selectors. The important fact is that Σ is what sometimes is called a resource. It means that, if in an article a selector is introduced, it is a new selector, the system knows that all selectors introduced previously are different. Let us identify the selectors with natural numbers, for a while. The Mizar system actually does this. However this identification is local. I mean that in other articles other natural numbers may be assigned to the same selector. This way we again get a permissiveness, perhaps more important. The identification with the natural numbers is allowed, but what one proves must not depend on the choice of the identification.

Selectors cannot be identified with selector symbols, of course. Mizar allows overloading of selectors. However, the Library Committee has a steady policy of elimination of overloading of selectors, so it rarely occurs in the MML.

Selectors are not introduced independently but in structures, and when a selector is introduced a type is assigned to it. The structure mode determines the (finite) set of selectors (new or inherited) and an assignment of types to the selectors. Please observe, that in these types the terms of the form **the** *Selector Symbols* may occur (without arguments) with the meaning: the field that corresponds to the selector in this structure object. Such terms are allowed only in structure definitions. In the definition of `lmult` (in `VectSpStr`) there is an example: it is of the type "a function from the Cartesian product of the carrier of F and the carrier to the carrier". It means that if F is a field and we deal with a linear space over F, then the domain of `lmult` is the Cartesian product of the scalars (the carrier of the field) and the vectors (the carrier of the linear space itself).

We are able now to define the denotation of a structure mode μ. Let $\Sigma_0 = \{\sigma_1, \ldots, \sigma_k\}$ be the set of selectors related to μ, and

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

will be assignment of types. The denotation of μ consists of all partial functions

$$f : \Sigma \dot{\rightarrow} \Omega$$

that enjoy the two conditions:

1. $\Sigma_0 \subseteq \mathrm{dom}(f)$
2. The types of objects assigned to selector are appropriate (what I cannot, because of technicalities, define here; I hope the meaning of it is clear).

Let us note that we do not require that

$$(**) \qquad \Sigma_0 = \mathrm{dom}(f)$$

In a previous version of Mizar such requirement was used. It caused obvious problems. The `VectSpStr`, if the requirement (**) was fulfilled, could not be widened to `GroupStr`, because the denotation of `GroupStr` consisted of functions with 4-element domains, and the denotation of `VectSpStr` of function with 5-element domains. In the semantics of Mizar used now this widening is allowed.

The attribute `strict` related to the structure μ, means that the requirement $(**)$ is fulfilled.

The forgetful functor **the** μ **of** A, if A denotes the function f, denotes the restriction $f|\Sigma_0$. Therefore the result is always strict.

The aggregating functor $\mu \ll A_1, \ldots, A_k \gg$ that assigns to the selector σ_i the object denoted by A_i, gives the function f explicitly. It is strict, too. (Sorry, this is not precise. The structure mode determines only the set of selectors. To an aggregating functor, a list of selectors is related, and the order of arguments in an aggregate must be the same as the order of fields in the structure definition. Because of this the inherited selectors must be repeated in the structure definition).

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. Introduction to lattice theory. *Formalized Mathematics*, 1(**1**):215–222, 1990.