Mizar Users Group

An Outline of PC Mizar

Michał Muzalewski

Series Editor Roman Matuszewski

Fondation

Philippe le Hodey

Brussels, 1993

All exercises are checked by the PC Mizar system

Translated from the Polish by Olgierd A. Wojtasiewicz.

Orders should be addressed to:

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

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

Contents

Introduction

Our objective is to learn to write Mizar articles. The word "Mizar" is ambiguous:

1. Mizar is language for the formalization of mathematics.

2. Mizar is a computer system connected with the processing of articles.

PC Mizar system is implemented by A. Trybulec and Cz. Byliński. Andrzej Trybulec is author of the Mizar language.

In *An Outline of PC Mizar* we shall discus successively the language and the system.

I am obliged to Andrzej Trybulec and Czesław Byliński, who took trouble of reading the preliminary version several times, which each time inspired me to further work and also enabled me to eliminate many errors. When writing this text I based myself on many works by A. Trybulec pertaining to Mizar, and also on his lectures dedicated to that language and that system.

My thanks are also due to Krzysztof Prażmowski, who read the typescript and shared with me his many valuable reflections intended to make my text easier to beginners in the application of Mizar.

Part One: The Language

1. Article

In a Mizar article, as in a standard mathematical article, we introduce new concepts, prove the correctness of their definitions, and prove theorems. The proofs of those theorems are based on theorems earlier proved by use in the article of refer to theorems proved in other articles or else are based directly on axioms formulated in the articles `TARSKI` and `AXIOMS` (which are checked by the system Mizar solely from the point of view of their linguistic correctness).

Every articles has the form:

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

The *environment* is a sequence of directives by means of which we draw from Mizar's actual library the information we need. On the basis of such information we can justify in the text the correctness of the definitions introduced and prove theorems. More on directives can be found in the chapter "Directives".

Further, certain words will be written in bold type. They will be words reserved for Mizar, that is such whose meanings are rigorously determined by definition in the Mizar language. That typographical distinction is to draw the Reader's attention to then, and thus more easily to remember at last some of them.

The text contains definitions, theorems, and their proofs. To put it a little more rigorously, the units of the text are reservations, definitions, redefinitions, schemes, end theorems, which in the Mizar grammar are termed statements.

It is worth while saying now that, from the point of view of the Mizar grammar, proofs, or - more broadly speaking - justifications are immanent units of definitions, redefinitions, schemes, and statements. In other words, the said grammatical units denote certain mathematical contents taken together with their justification.

Consider, by way of example, the meaning of the word *definition*. The content of a *definition* consists of
definiendum (the expression to be defined) and *definiens* (the defining expression).
Hence *definition*: *definiendum* *definiens* *proof of correctness*
Likewise, *statement* means both a certain and its justification.

An *abstract* is another form of an article. It is obtained from a Mizar article by the elimination of all proofs, and also all lemmas and definitions of private objects. Only public definitions and theorems are left. It is just to the abstract that we look when availing ourselves of other articles. Only that which has been transferred to the *abstract* from a Mizar article may be used in successive articles.

Basic Linguistic Constructions

2. Types

In Mizar, every variables has its type. Some knowledge of the content of the Main Mizar Library is required to know which types can be used.

New types are introduced by definitions of structure and definitions of mode.

The type of a variable may be fixed globally for the entire article by reservation, for instance

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

When formulating a sentence with the use of a variable whose type has not been reserved earlier we must explicitly state its type, for instance:

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

`X` is variable qualified implicitly: its type has been reserved earlier; `a` is a variable qualified explicitly, has the type `Any` given in the sentence.

Types may have *arguments*, i.e., be derived from other objects. The symbol of the type and its arguments are separated by the word `of`, e.g.,

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

We shall now briefly discuss the basic types in Mizar.

The basic types in Mizar are to be found in the article `HIDDEN`, which is automatically joined to every article. Those automatically joined types are called built-in types. The article `HIDDEN` describes the following types:

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

where `X` has the type `set`, and `D` has the type `DOMAIN`.

The type `Any` is the broadest type in Mizar; every other type expands to the type `Any`.

The type `set` is equal to the type `Any`.

The type `DOMAIN` denotes non-empty sets, that is so-called domains; `SUBDOMAIN of D` (subdomains - subsets which are domains), `Real` (real numbers), `Nat` (natural numbers).

The use of the type θ introduced in another article α must be preceded by two directives included in the environment:

`vocabulary` α`;` `signature` α`;`

The above does not apply to built-in types, that is those which have been introduced in the article `HIDDEN`.

3. Terms

In the Mizar language, as in standard mathematics, functions are defined. To put it more precisely, what is defined is the names of functions, that is functors. Symbols of functors are used in the construction of terms. In mathematics, terms are expressions received from the iteration of functor symbols: variables and constants are terms, and moreover if τ_1, ... , τ_p are terms, and φ is a symbol of functor of p arguments, then φ (τ_1, ... , τ_p) is also a term.

Thus, for instance, if `x,y` are variables, and `+` is a symbol of a binary functor, then:

`x,y` are terms of degree 0,

`x + y, y + x` are terms of degree 1,

`x + (x + y), (x + x) + (x + x)` are terms of degree 2, etc.

In the Mizar language the concept of term is interpreted more broadly. For instance, the grammar of Mizar admits the term `it` which denotes within a definiens of mode or functor (that term may not be used outside a definiens) the object being defined. The sentences recorded with the use of `it` are simply shorter and more legible.

We shall confine ourselves to indicating the five most important kinds of terms:

- identifiers of a variable,
- terms with a public functor,
- terms with a private functor,
- terms with the Frænkel operator,
- aggregates.

The arguments of the terms with a public functor may be written:

- on the left side of the functor symbol,
- on the right side of the functor symbol,
- on both side of the functor symbol simultaneously.

If the number of the arguments on one side of the functor symbol is two or more, then the arguments must be separated by commas, and the list of the arguments must be placed within round brackets:

φ, `x`φ, φ`x`, `x`φ`y`, `(`x_{i1}`, ... , `x_{ik}`)`φ`(`x_{j1}`, ... , `x_{jl}`)`

where φ is the functor symbol.

The arguments of the terms with a private functor may be written only on the right side, and the (round) brackets are obligatory even if the number of arguments equals 0:

`F( )`, `F(x)`, `F(`x_1`, ... , `x_n`)`

where `F` is the identifier of a private functor.

Let us revert to the terms with a public functor. As has been said, the arguments of such a term are written on one side of the functor symbol (on the left - prefix recording, on the right - postfix recording) or on both sides simultaneously (infix recording). Thus

`bool X`, $\cup$ `X`, `-1`, `graph f`, `dom f`, `rng f`, `id X`, `"f`, `Funcs(X,Y)`,
χ`(A,X)`, `len p`, `Plane(A,B,C)`

are prefix terms,

`x"`, `x*`, `x!`

are postfix terms, and

`x + y`, `f|X`, `f"X`, `f.(a,b)`

are infix terms.

Let it be sad once more:

- a single argument may, but need not, be bracketed,
- two or more argument must be bracketed.

For instance: one may write `(x)"`, but `x"` is enough, on the other hand the inscription `Funcs X,Y` instead of `Funcs(X,Y)` is not allowed.

The above discussed terms with a public functor consisted of a functor symbol and arguments. In Mizar also such terms are allowed in which next to the functor and its arguments also so-called functor brackets occur. The symbols `[` and `]`, { and }, and `[:` and `:]` are examples of left and right functor brackets. And here are mathematical objects and terms describing those objects and built with the use of the said bracket:

1. ordered pairs, triples, quadruples, and further n-tuples up to ordered nonatuples inclusive:
   `[x1,x2]`, ... ,`[x1,x2,x3,x4,x5,x6,x7,x8,x9]`.
2. singleton, pair, and further finite sets, until the set of eight elements included:
   {`x1`}, {`x1,x2`}, {`x1,x2,x3`}, ... , {`x1,x2,x3,x4,x5,x6,x7,x8`}.
3. the Cartesian product of sets:
   `[:X1,X2:]`, `[:X1,X2,X3:]`, `[:X1,X2,X3,X4:]`.

The simplest Frænkel term is an expression of the form

{ *term* : *sentence* }

And here is an example of a Frænkel term:

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

The variable `p` in this case denotes an ordered pair of points, and `p~` is the equivalence class of the element `p`. Thus the above term denotes the set of all free vectors of the structure `M`.

To conclude let us formulate the principle of the identification of functors, that is reply to the question how functors are distinguished. Now the system Mizar identifiers a functor by

- symbol,
- number of arguments,
- types of arguments.

Thus if two functors differ from one another by a symbol, the number of arguments, or type be it of single argument only, the system treats those functors are different.

4. Atomic formulas

In the Mizar language, as in standard mathematics, relations are defined. To put it more precisely, what is defined is the names of relations, that is predicates. Predicates are used to construct atomic formulas, and these in turn to construct - with the use of connectives and quantifiers - arbitrary formulas.

For the time being we shall discuss atomic formulas. The basic kinds of atomic formulas are:

- atomic formulas with a public predicate,
- atomic formulas with a private predicate,

In the former case arguments may be written:

- on the left side of the predicate symbol,

- on the right side of the predicate symbol, or
- on both side of the predicate symbol simultaneously.

The list of arguments is not bracketed:

Π, xΠ, Πx, xΠy, x_{i1} , ... , $x_{ik}\Pi x_{j1}$, ... , x_{jl}

where Π stands for the symbol of the predicate.

In particular, atomic formulas with a public predicate include the equational formulas:

`x = y` and `x <> y` .

In the latter case (that of atomic formulas with a private predicate) arguments may be written solely on the right side, and (square) brackets are obligatory even if the number of arguments is 0:

`P[ ]`, `P[x]`, `P[`x_1 , ... , x_n`]`

where `P` stands for the identifier of a private predicate.

Further the atomic formulas include the qualifying formulas, that is expression of the form *term* `is` *type* , for instance:

$\sqrt{}$`(`b^2 `- 4ac) is Real`

`GroupsStr` ≪ `REAL, addreal, compreal,0` ≫ `is AbGroup`

As in the case of functors, the system identifiers a given predicate by means of a symbol, the number of arguments, and types of arguments. Any two predicates which differ form one another by the symbol, the number of arguments, or the type be it of only one argument are treated by the system as different predicates.

5. Formulas

Negation, conjunction, disjunction, implication, and equivalence are denoted, respectively, by `not`, `&`, `or`, `implies`, `iff`.

The above order corresponds to the decreasing strength of binding by those connectives, with the proviso that the strength of binding of implication is the same as that of equivalence.

Associative recording is admissible for conjunction and disjunction, which is to say that the system itself adds the missing brackets. For instance the formula

ϕ_1 `&` ϕ_2 `&` ϕ_3

is transformed by the system into

`(`ϕ_1 `&` ϕ_2`)` `&` ϕ_3

On the contrary, for `implies` and `iff` the associative recording is not admissible, which is to say that, for instance, the expression

ϕ_1 `iff` ϕ_2 `iff` ϕ_3

is treated as erroneous.

There are three kinds of quantified sentences:

1. a universal simple formula, e.g.,
   ```
   for x being Any holds x = x;
   ```
2. a universal limited formula, e.g.,
   ```
   for x being Real st x <> 0 holds x·x" = 1;
   ```
3. an existential formula, e.g.,
   ```
   ex x, y st x <> y;
   ```

The word `holds` preceding a quantified formula may be omitted, and hence, for instance,

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

may be replaced by:

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

The User is warned that in Mizar quantifiers bind less strongly than connectives do. Thus

`for x holds` α `&` β

means the same as

`for x holds (`α `&` β`)`

but not

`(for x holds` α`) &` β .

Proving of Theorems

6. Simple justification

We shall discuss in turn the various kinds of simple justifications.

1. When referring to sentences previously assumed or justified we write

by ϱ_1 , ... , ϱ_n ;

where ϱ_1 , ... , ϱ_n are references.

EXAMPLE

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

where

- the private reference `A` is the label of a sentence which has been proved earlier in this article,
- the library reference `TARSKI:2` indicates the second theorem in the article `TARSKI`,
- the library reference `BOOLE:def 1` indicates the first definitional theorem in the article `BOOLE`.

2. When referring to a schema we write

from *identifier-of-schema* (ϱ_1 , ... , ϱ_n);

where ϱ_1 , ... , ϱ_n are references, if the schema has premises, or

from *identifier-of-schema* ;

if the schema has no premises.

EXAMPLES

```
ex Z st x holds x ∈ X iff x ∈ X & x ∈ Y from Separation;
```

```
thus thesis from FuncEx (A1,A2);
```

The words `Separation` and `FuncEx` are identifiers of schemes, while `A1, A2` are labels of the sentences which are premises, that is assumptions of the schema `FuncEx`. Those sentences must be proved first in order to justify on the basis of that schema. On the contrary, the schema `Separation` has no premises.

3. If the sentence written out is obvious to CHECKER (see *14. Basic informations of Checker*) then the entire justification may be written in the form of the semicolon, e.g.,

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

In Mizar, the symbol `c=` denotes the inclusion of sets.

4. When we refer the sentence preceding a given sentence it suffices to write `then`.

Thus instead of

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

we may write more simply:

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

The expression `then` is termed linking. Owing to linking we have avoided labeling the second sentence.

7. Formula "thesis", skeleton of the proof, and results of reasoning

The word `thesis` means *that which remains to be proved.*
Self-evidently, `thesis` at the beginning of a proof just means the thesis of that proof. Since Mizar admits proofs within proofs, that is nested proofs, the thesis of the proof will sometimes be called the main thesis. Thus the main thesis is the initial value of the dynamically changing formula `thesis`. The changing value of `thesis` is computed as we read the text from `proof` to `end`.

1. If we prove the implication

   α `implies` β

   and assume its antecedent

   `assume` α`;`

   then `thesis` denotes its consequent β.

2. If, when proving the conjunction

   α `&` β

   we write down the conclusion

   `thus` α`;`

   then `thesis` denotes the sentence β.

3. If we prove the universal sentence

   `for x being` θ `holds` β

   and write out the generalization

   `let x be` θ`;`

   then the word `thesis` denotes β.

4. If we prove the existential sentence

$$\texttt{ex x being } \theta \texttt{ st } \beta$$

we write the exemplification

$$\texttt{take x;}$$

the `thesis` denotes β.

These four constructions:

- assumption (1),
- conclusion (2),
- generalization (3), and
- exemplification (4).

from the record of the skeleton of the proof. They, as it were, cut off from the main thesis its initial fragments and correspondingly modify the value of the formula `thesis`:

1'. The assumption `assume` α`;`cuts off expression

$$\alpha \texttt{ implies}$$

from the sentence

$$\alpha \texttt{ implies } \beta$$

leaving the expression β.

2'. The conclusion `thus` α`;` cuts off expression

$$\alpha \texttt{ \&}$$

from the conjunction

$$\alpha \texttt{ \& } \beta$$

leaving the expression β.

3'. The generalization `let x be` θ`;` cuts off the expression

$$\texttt{for x being } \theta \texttt{ holds}$$

from the universal sentence

$$\texttt{for x being } \theta \texttt{ holds } \beta$$

leaving the expression β.

4'. The exemplification `take x;` cuts off the expression

$$\texttt{ex x being } \theta \texttt{ st}$$

from the existential sentence

$$\texttt{ex x being } \theta \texttt{ st } \beta$$

leaving the expression β.

EXAMPLE
Let us analyse the following fragment of a Mizar article:

```
reserve x,y for Any;
theorem T23:  (ex x for y holds α) implies (for y ex x st α)
       ::thesis = (ex x for y holds α) implies (for y ex x st α)
proof
  assume ex x for y holds α;
                ::thesis = for y ex x st α
  then consider y such that E: for y holds α;
                ::thesis = for y ex x st α
  let y;
                ::thesis = ex x st α
  E': α by E;
                ::thesis = ex x st α
  thus thesis by E';
                ::thesis = VERUM
end;
```

(from :: until *end–of–line* - comments in Mizar text - not processed by PC Mizar system).
The sentence

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

states the assumed existential statement

`ex x for y holds` α `;`

yields the statement of the choice

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

That statement chooses such a value of the variable `x` for which the sentence `E` holds.

In the above recording of our reasoning next to the assumption

`assume ex x for y holds` α`;`

the generalization

`let y;`

and the conclusion

`thus thesis by E';`

which form the skeleton of the proof and modify `thesis`, there also occur the statement of choice

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

and the labelled sentence

`E':` α `by E;`

which do not modify `thesis` (they constitute the so-called auxiliary recording of the recording of the reasoning).

It is self-evident that when `end` appears in the proof the dynamic formula of `thesis` should be a VERUM (precisely speaking: `not contradiction`). Otherwise Mizar reports an error of REASONER #70 *(Something remains to be proved).*

Suppose now that we read the proof backward end the conclusion `thus` β`;` has, for simplicity, the open form (the open form means `thus thesis`). As we read from `end` to `proof` we dynamically reconstruct the changing result of the reasoning by writing to the left of the sentence β the appropriate expressions:

1". The assumption `assume` α`;` adds to the sentence β the expression α `implies` thus forming the expression α `implies` β.

2". The conclusion `thus` α`;` adds to the sentence β the expression α `&` thus forming the expression α `&` β.

3". The generalization `let x be` θ`;` adds to the sentence β the expression `for x being` θ `holds` thus forming the expression `for x being` θ `holds` β.

4". The exemplification `take x;` adds to the sentence β the expression `ex x being` θ `st` thus forming the expression `ex x being` θ `st` β.

EXAMPLE

Suppose that we have found a sheet of paper with the following proof:

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

Unfortunately the thesis being proved was not recorded on that sheet. When reading the proof backward we easily find the result of the reasoning, which obviously is the sentence

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

The only thing which we have not succeeded in reconstructing is the type of the variable y because it had been fixed in the reservation preceding the statement the proof of which we have found on the said sheet.

The various methods of skeletoning proofs and the practical methods of computing the formula `thesis` will be explained below in greater detail by reference to examples which will help us to discuss the various tactics of proving theorems. On the other hand, the problem of computing the result of the reasoning will reappear a little later in connection with the discussion of diffuse statements, which consist in that their thesis is not open.

EXAMPLE

To conclude this chapter we give the shortest statement:

```
not contradiction proof end;
```

The word `contradiction` denotes the logical constant *falsehood*. Obviously, `not contradiction` denotes the logical constant *truth*. Note also that in the Mizar language `contradiction` and `not contradiction` are interpreted as sentences.

8. The tactics of proving theorems

The fundamental principle in proving theorems is that we do not write out at once the complete reasoning but only its skeleton, that is

- assumption, and
- conclusion, and also
- generalizations, if we prove universal sentences, and
- exemplifications, if we prove existential sentences.

The beginners may find it considerably difficult to skeleton the proofs. We shall present many methods of skeletoning, even though not all of them. We shall disregard

proofs by definitional expansion and proofs *per cases*. To simplify natters we shall also disregard simple justifications, that is justification by `by` and by `from`. Such a method of constructing proofs is also natural in so far as it is recommended to add simple justifications to already verified skeletons. We begin the description.

At first we have to formulate the thesis of our proof. Then we write

`proof` ... `end;`

The dots will, of course, be replaced by the successive steps of the reasoning. The last step is often

`thus thesis;`

The steps of the reasoning are separated by semicolons. What those steps are depends self-evidently on whether the proof is direct, or by contradiction, and on the thesis of the proof.

Proving a thesis α by contradiction reduces to the proof of the implication

`not` α `implies contradiction`

Those Readers who are familiar with *Jaśkowski's proofs* by assumption will note the convergence of the rules used in Mizar and those in *Jaśkowski's system*. That applies in particular to proving by contradiction.

We shall now successively examine the following forms of the thesis:

1. the thesis is a conjunction,
2. the thesis is an implication,
3. the thesis is an equivalence,
4. the thesis is a disjunction,
5. the thesis is a universal sentence,
6. the thesis is an existential sentence.

We shall discuss one by one the tactics of writing proofs in the cases from 1 to 6.

8.1. Proving a conjunction

In the proof we have to state the truth of every constituent of the conjunction. Thus, if the thesis is in the form

$$C_1 \ \& \ C_2 \ \& \ C_3$$

then the proof will be as follows:

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

The word `thus` indicates that the sentence that follow is a *conclusion*. By *conclusion* we mean the thesis of the proof or its part. The dots indicate simple justification or proof. For instance:

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

The word `thesis` denotes here the third constituent of the conjunction being proved, that is the sentence `for x holds x ∈ X ∩ Y iff x ∈ X & x ∈ Y`.

8.2. Proving an implication

Tactics 1 - the direct method

The first step of the reasoning is

`assume` *the-antecedent-of-the-implication* ;

and the last

`thus` *the-consequent-of-the-implication* ;

The word `assume` points to the assumption made in the proof. The assumption may be a single sentence (the so-called *single assumption*) or a sequence of labelled sentences linked together by the word `and` (the so-called *collective assumption*). In the latter case note that the splitting of the assumption may be convenient because it makes it possible to refer to partial assumption of the collective one.

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

The formula `thesis` denotes here the sentence `X ∩ Y <> ∅` . In this case the replacement of the *single assumption*

```
assume x ∈ X & x ∈ Y;
```

by the *collective assumption*

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

is not recommended because after the *collective assumption* there is no linking (see *10. Conventions*).

Tactics 2 - the indirect method

The first two steps are the assumptions:

`assume` *the-antecedent-of-the-implication* ;
`assume not` *the-consequent-of-the-implication* ;

The last step is the expression

```
thus contradiction;
```

The following proof is incorrect:

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

Why is that so?

Mizar transforms the thesis

α implies β

into the sentence

not (α & notβ)

and expects a proof of that sentence precisely. But we have proved the sentence

not β implies not α

which is transformed by Mizar into the sentence

not (not β & α)

In the algebra of semantic correlates used by REASONER, that is the module of the system which checks the correctness of the skeleton of the proof, conjunction is not commutative, and this is why the system will report an error, which indicates a lack of sufficient justification. A more precise explanation would require a deeper insight into the semantics of Mizar.

The above proof can be *saved* as follows:

Tactics 3 - diffuse statement

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

The construction

now assume not β; ... thus not α; end;

is called a *diffuse statement* (currently: proof by now). This statement is marked by the fact that its thesis is not explicitly written down, but REASONER (the module of the processor responsible for the skeleton of the proof) itself reconstructs the thesis of the diffuse statement. The above diffuse statement has the following content:

not β implies not α

Mizar (or, to put it more precisely, REASONER) expects the sentence

α implies β

but the use of the linking hence thesis after the diffuse statement is a direct justification of the theorem being proved.

In fact, CHECKER (the module of the processor responsible for direct justifications) by referring to a tautology of the propositional calculus (which it may always do "by itself") will easily transform the proved diffuse statement into the main thesis.

8.3. Proving an equivalence

Tactics 1

If the thesis is an equivalence

α `iff` β

then we have to prove two implications

α `implies` β and β `implies` α

where the order in which these implication are listed is essential and must be such as above. Thus the proof of the thesis is as follows:

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

The word `thesis` self-evidently denotes here the sentence β `implies` α.

Tactics 2 - two diffuse statements

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

In the above proof the order in which the diffuse statements with theses `A` and `B` are proved is inessential because CHECKER, when verifying the direct justification `thus thesis by A,B` may make use of the tautologies of the propositional calculus. Note also that the label `B:` is superfluous because we may refer to the thesis of the second diffuse statement by the linking `hence thesis by A`.

8.4. Proving a disjunction

Tactics 1

If the thesis is in the form

D_1 `or` D_2

then it is convenient to assume the negation of one constituent of the disjunction and to prove the other. In such a case the proof takes on the following form:

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

For instance

```
{x} \ X = ∅ or {x} \ X = {x}
 proof
   assume {x} \ X <> ∅;
```

```
    then B: not x ∈ X by Th24;
    thus thesis by B,Th22;
  end;
```

The word `thesis` denotes here the sentence {x} \ X = {x}. Here, too, the label is superfluous; one can refer to the sentence with the label `B:` by the linking `hence thesis by Th22`.

Tactics 2 - replacement of disjunction by two implications

Let γ be an auxiliary sentence

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

The label `B:` may be eliminated if the linking `hence thesis by B` is used.

Tactics 3 - indirect

```
α or β
 proof
  asume that A: not α and B: not β;
  ..........
  thus contradiction;
 end;
```

8.5. Proving an universal sentence

When we want to prove a universal statement we consider any object of a given type. In Mizar we proceed likewise.

The proof of a sentence in the form

`for x being θ holds ... ;`

then begins with the so-called generalization, that is the expression

`let x be θ;`

If the sentence being proved is in the form

`for x being θ st ω holds ... ;`

then the first step in the proof will be

`let x be θ such that L: ω;`

EXAMPLE

```
for X,Y,Z being set holds X c= Y & Y c= Z implies X c= Z
 proof
   let X,Y,Z be set;
   assume A: X c= Y;
   assume B: Y c= Z;
   C: for x holds x ∈ X implies x ∈ Z
    proof
```

```
      let x such that D: x ∈ X;
      E: x ∈ Y by A,D,BOOLE:5;
      thus x ∈ Z by B,E,BOOLE:5;
     end;
    thus X c= Z by C,BOOLE:5;
  end;
```

The same proof may be recorded in a simpler way:

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

In fact:

a) The sentences

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

are treated by the system as equivalent. When passing to semantic correlates the system adds the missing quantifiers.

b) After the *cutting off* of the sentences `A:` and `B:` from the main thesis the word `thesis` in the last but one step of the proof denotes the consequent only, that is the sentence `X c= Z` which is not written explicitly.

8.6. Proving an existential sentence

The proof of the existential sentence

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

consists in indicating an object which satisfies the condition `W`. The relevant proof is thus as follows:

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

The symbol `c` denotes the object in question. The expression `take x = c;` indicates that objects. But how is that object to be found?

Two methods of finding such an object can be indicated.

The first consists in the construction of the constant c from the constants of the theory under consideration, if that theory has some constants or makes it possible to define them. This method might be termed effective.

The second, non-effective, consists in availing oneself from an existential sentence which has been proved earlier or assumed.

EXAMPLE of the effective proof:

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

The word `thesis` denotes here the sentence `0 <> 1 & 0 <> 2 & 1 <> 2` , which is obvious to Mizar.

EXAMPLE of the non-effective proof:
Let Π stand for symbol of any binary predicate.

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

The label `A:` cannot be eliminated because after `consider` linking is not allowed. The word `thesis` in its first occurrence denotes the sentence `for y ex x st xΠy` because the assumption cut off the antecedent of the implication from the main thesis. The word `thesis` in its second occurrence (in the nested proof) denotes, after the generalization `let y`, the sentence `ex x st xΠy`. That sentence follows by linking from the sentence `B:` because CHECKER applies the rule called the law of abstraction from concreteness (see *14. Basic informations of Checker* below).

EXAMPLE as a warning!
Let us read carefully the following two sentences:

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

Their equivalence in the predicate calculus self-evidently does not give rise to any doubt. The brackets in the second sentence are superfluous, but they had better to be retained because the alarmed Reader could look for the solution of the puzzle in a wrong place. And the puzzle is: are these two sentences equivalent from Mizar's point of view? Now they are not. When proving the first sentence we use the exemplification `take x;`, where `x` has the type θ, and we prove the sentence α. When proving the second sentence we use the exemplification `take x;` where `x` has the type `Any`, and we prove two sentences: `x is` θ and α.

9. Change of type

In practice it is often necessary to change the type of the objects occurring in the reasoning. This is achieved by the construction `reconsider`, i.e., the statement of a change of type, which has the form:

`reconsider x =` τ `as` θ `by ... ;`

or

`reconsider x =` τ `as` θ `from ... ;`

where τ is a certain term.

The construction `reconsider` introduces a new variable `x` which denotes the same as τ but has the type θ. The statement of a change of type is justified by the qualifying formula:

τ `is` θ

EXAMPLE

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

By means of `reconsider` we change the type locally, that is within the reasoning. A global change is achieved by redefinition (see *25. Redefinitions*).

Suppose that the term τ is the variable `x` and has the type θ. The *statement of a change of type* may be recorded thus:

`reconsider x as` θ' `by ... ;`

or

`reconsider x as` θ' `from ... ;`

The name *statement of a change type*, which is to be used in accordnace with the Mizar grammar, may in this case be misleading because the "working" of that statement is such that the variable `x` acquires the new type θ' without losing its old type θ. We thus have to do in fact not with the change of one type into another but with the expansion of the set of the types ascribed to the given variable. The operation of *expansion of the set of types*, presented above, "works" only on variables; it does not work on arbitrary terms.

10. Conventions

10.1. Reservation

Reservation makes it possible to shorten formulas. The expression

`reserve x for` θ`;`

establishes that the variable `x` has type θ in those occurrences in which the type of the variable `x` is not stated explicitly.

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

In the sentence `A:` the variable `x` has the type `set` stated in the reservation (the variable `x` is a variable qualified implicitly). In the sentence `B:` the variable `x` has the type `Any`

(here `x` is qualified explicitly). The ascribing to `x` of the type `Any` in the sentence `B:` does not cancel the initial reservation. In fact, in the successive sentence

```
C: ex x st x = ∅;
```

the system will identify `x` as a variable of the type `set`.

10.2. Linking by `then`

The preceding of a sentence Z by the word `then` means that in the justification of Z we avail ourselves of the sentence `A:` which immediately precedes the sentence Z. This way of justifying is called linking.

It must be emphasized that linking requires that the expression `A:` be a sentence. It is thus not allowed to use linking after `proof`. Nor is it allowed to use linking after a collective assumption for then it could not be clear to which of the partial assumptions we refer. Finally, it is not allowed to use linking after those constructions which - as we put it - do not leave a sentence:

- after the generalization `let` ... ;
- after the exemplification `take` ... ;
- after the statement of choice `consider` ... ;
- after the existential assumption `given` ... ;
- after the statement of a change of type `reconsider` ... ;

After the conclusion `thus` ... ; and `hence` ... ; linking is admissible.

10.3. Linking by `hence`

If one of the premises of the conclusion is the sentence which precedes that conclusion then one can use linking by replacing `thus` by `hence`. Figuratively speaking

`then` and `thus` is equal `hence`

10.4. Existential assumption

The construction

`given` ... `such that` ... ;

is called *existential assumption*. It replaces the assumption on the existence of certain objects (`assume ex` ...) combined with the choice of one of those objects (`consider` ...). Linking after an existential assumption is not allowed.

EXAMPLE
Suppose that the statement expresses the commutativity of multiplication.

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

11. Nested proofs

Mizar admits the construction of proofs within other proofs, called nested proofs. The rules of the construction of inner proofs are the same as those for main proofs.

12. Diffuse statement

It sometimes occurs that we prove a theorem which has not been formulated openly. In Mizar, such a construction is called a *diffuse statement* and we begin it with the word `now`. It looks thus:

`L: now` ... `thus` β`; end;`

The word `now` is followed by the proof of the conclusion β (the word `proof` is not written after `now`).

EXAMPLE

`L: now assume` α`;` ... `take x; thus` β`; end;`

The direct justification `by L` is consists in the reference to the sentence

α `implies ex x st` β

REMARK 1

The conclusion β must be written out openly and may not be replaced by the word `thesis`. We shall explain why.

The reference to the label `L:` is a reference to a thesis proved by the construction `now`, or, to put it more precisely, to the result of such a reasoning. Of course, if the result of that reasoning is to be computable β must be formulated openly.

REMARK 2

It is not allowed to precede `now` by the symbols `then`, `thus`, and `hence`.

To conclude these brief comments on diffuse statements we shall try to make the Reader sensitive to a certain danger connected with the use of that construction. Please, examine the following reasoning:

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

The above reasoning is incorrect. Under the label `A:` there is the sentence

`x < y implies` γ`;`

In that sentence `x,y` are constants. Hence from that sentence it does not follow that

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

Hence the sentence

```
y < x implies γ;
```

does not follow either. Thus the justification of the conclusion

```
hence γ by A;
```

is erroneous. And here is the corrected version of the above proof:

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

The sentence `A:` is the following universal sentence:

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

and that is why it is applicable in the conclusion below:

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

13. Iterative equality

Iterative equality is the construction of the following form:

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

where τ_0, ... , τ_1 are terms.

EXAMPLE

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

The content of iterative equality is the equality of the first and the last term occurring in that construction.

14. Basic informations of Checker

1. Every tautology of the propositional calculus is accepted by CHECKER without justification.

2. A sentence beginning with a quantifier requires justification.

3. When verifying the correctness of a justification CHECKER takes into consideration only the following premises:
   - those specified after `by` or `from`,
   - those resulting from linking (`then`, `hence`).
4. The relation of equality is treated by CHECKER as reflexive, symmetrical, transitive, and extensional. For instance, owing to extensionality CHECKER accepts the following sentence
   `x = y & x <> ∅ implies y <> ∅;`
5. CHECKER usually does not accept a justification which among its premises has more than one universal sentence. CHECKER will, however, accept the justification
   C_1 `&` C_2 `by` O_1`,` O_2 `;`
   if O_1 is the justification of C_1 and O_2 is the justification of C_2.
6. CHECKER automatically, i.e., without reference to justification, applies the following two rules of inference of the classical functional calculus:

R1: The law of abstraction from concreteness,
R2: The law of transition from the general to the particular.

If `c` stands for a constant of the type θ, then the above rules can be recorded in Mizar as follows:

rule R1:

$$\frac{\texttt{W(c);}}{\texttt{then ex x being } \theta \texttt{ st W(x);}}$$

rule R2:

$$\frac{\texttt{for x being } \theta \texttt{ holds W(x);}}{\texttt{then W(x);}}$$

15. Schemes

Schemes are sentences of second order. First comes the reserved word `scheme` and the identifier of the schema. They are followed by the list (in square brackets) of the parameters of the schema (functors or predicates) which play in the schema the role of variables of second order. Next come, after the colon, successively the thesis of the schema and possibly the premises (that is, the assumptions of the schema). Finally comes the justification of the schema.

Here are examples of schemes (with justifications replaced by dots), namely `Separation`, without premises, and `FuncEx`, with two premises.

Let `F` stand for the identifier of a private functor (see *3. Terms*), and `P`, for the identifier of a private predicate (see *4. Atomic formulas*).

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


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

The premises `A1:`, `A2:` are preceded by the word `provided` and separated from one another by the separator `and`.

Finally one more example perfectly well known to secondary school pupils, namely the schema of induction:

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

The Reader has probably noted that the scope of the applications of a given schema is determined by the parameters of the latter. For instance, the schema of induction allows one to prove sentences of the form

`for k being Nat holds` ϕ`(k)`

where ϕ is any sentence with one free variable `k`. The justification of that sentence is a follows. We first prove the sentences:

`C1:` ϕ`(0);`

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

These sentences are premises which allow us to avail ourselves of the schema `Ind:`

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

The schema of induction has been proved in the article `NAT_1`. In order to make use of that schema one must join it to the environment, which is to say that one has to write the directive `schemes NAT_1` between the words `environ` and `begin`.

Definitions

16. Review of definitions

In the Mizar language, as in standard mathematics, we define relations (strictly speaking, names of relations, that is predicates) and functions (strictly speaking, names of functions, that is functors). Moreover, in Mizar, more explicitly than it is usually done in mathematics, we use definitions to introduce structures (such as the structure of group), used in turn to define other objects. In our example they will be groups. On the other hand, the peculiarity of Mizar consists in types, which are defined, too: Mizar is based on an system of types.

The definitions of predicates and structures do not require the verification of their correctness. The definitions of functors and types require a proof of the condition of existence, and those of functors also a proof of the condition of uniqueness.

In the Mizar language the basic objects are types, predicates, and functors. Some of those objects - let us call them primary - have been built-in in the system. Their full description is to be found in the articles `HIDDEN`, `TARSKI`, and `AXIOMS`. The review of the built-in types is to be found in the chapter on *Types*.

Secondary objects are introduced by definitions.

The defined objects are classed into private and public.

Private definitions (of a variable, a private predicate, a public functor) introduce useful notations, but their scope is local, that is limited to a single article. In the present text we shall discuss only the definition of variable.

17. Definition of variable

The construction

`set x =` τ`;`

where `x` is the identifier of a variable, and τ is a term, introduces the variable `x` as the designation of the term τ.

As the example we shall give the beginning of a theorem which begins with the definition of the variable designated by the identifier `X`. The Reader may look for further occurrences of that variable.

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

18. Public definitions

We now proceed to discuss public definitions, from definitions of types to those of predicate and functors.

Types are introduced by:

- definition of structure,

- definition of mode,
- definition with `is`, which forms abbreviations of existing types,
- registration of clusters.

These four kinds of definitions of types will be discussed successively.

19. Definition of structure

In Mizar, structure are finite sequences of objects of a fixed type, called *selectors of a given structure*. The definition of a structure does not require any conditions and correctness. In particular, the condition of existence need not be justified because the Mizar language.

Structures may be classed into:

- structures without parameter, and
- structures with parameter.

The structure without parameter are defined thus:

`struct` $\Sigma \ll \sigma_1 -> \theta_1$, ... , $\sigma_p -> \theta_p \gg$;

where

Σ is the symbol of structure,
σ_1 , ... , σ_p are symbols of selectors, and
θ_1 , ... , θ_p are types.

The symbol of structure and those of selectors should be included in the vocabulary and preceded by the classifiers `G` and `U`, respectively.

EXAMPLE 1

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

It is convenient to follow the definition of a structure with a series of definitions which facilitate the "service" of the structure defined. The author of the article from which the above example is drawn introduced, after the definition of structure, the following reservation:

```
reserve S for IncStr;
```

Next he defined the modes:

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

and the public predicates:

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

which correspond to the selectors `Inc1`, `Inc2`, `Inc3` for the structure `S` fixed by the reservation. Usually a structure is defined only in order to become the mother type of the mode of the structures which satisfy the axioms of a given theory. In the example under consideration, drawn from the article `INCSP_1` *(Axioms of Incidency)*, they are incidency spaces `IncSpace`. The Reader is advised to consult that article in order to become acquainted in the way in which that well-known fragment of geometry is rendered in the Mizar language.

The above example was that of a definition of a structure without a parameter. There are situations in which it is more convenient to make use of a definition of a structure with a parameter. Structures with a parameter are defined as follows.

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

where x_{i1}, ... , $x_{ik} \in \{x_1, \ldots, x_n\}$, and the remaining variables are reconstructible (see the example of the superposition of permutation in the chapter on *25. Redefinitions*).

EXAMPLE 2

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

After the symbol of structure (in this case: `VectSpStr`) we write the reserved word `over`, followed by the parameters (in the case under consideration there is only one parameter, namely `F`).

20. Definition of mode

Here is the form of the most often used definition of mode:

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

? is the symbol of mode and should be included in the vocabulary, preceded by the classifier `M`.
NOTE: We shall often replace recurrent constructions by dots. In the case under consideration the justification after the word `existence` was thus replaced.

In the above definition of mode only the condition of existence, specified by the word `existence`, is required. The condition of existence should include a justification of the following theorem:

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

And here is the form of a definitional theorem:

```
        for x1 being θ1 , ... ,xn being θn , y being θ(x1 , ... ,xn) holds
        y is ? of xi1 , ... ,xik iff δ(x1 , ... ,xn ,y);
```

EXAMPLE

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

The word `Function` is the symbol which denotes the type being introduced. The type function expands to the type `set`, or, if you prefer it that way, the type `set` is the mother type of the type function. The word `means` is followed by the definiens with the label `:FUNC:`. The object `it` of the type `set` has the type function if it is a set of ordered pairs

```
for p st p ∈ it ex x,y st [x,y] = p;
```

and meets the condition of uniqueness for the second element of the pair

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

The word `existence` is followed by the justification of the non-emptiness of the type being defined. The system expects a justification of the following sentence:

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

In the justification (see *8. The tactics of proving theorems* and *Proving an existential sentence*) it suffices to take empty set as an object of the type `set`:

```
take ∅;
```

The above definition has its analogue in the following definitional theorem:

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

Definitional theorems are automatically generated by the system. In order to avail oneself of a definitional theorem one has to find its number in the abstract of the article concerned.

21. Abbreviations (definitions with "is")

We shall now describe a very convenient method of defining modes which are extensionally equal to other modes:

`definition let` x_1 `be` θ_1 `,` ... `,` x_n `be` θ_n`;`
`  mode` ? `of` x_{i1} `,` ... `,`x_{ik} `is` θ`(`x_1 `,` ... `,` x_n`);`
`end;`

? is the symbol of mode and should be included in the vocabulary preceded by the classifier `M`.

The above definition does not include any definiens and hence, practically, it will play the role of a construction which enables one conveniently to introduce abbreviations.

The definition of mode by `is` does not require conditions of correctness.

The system, when processing a given article, replaces the abbreviations by the original type. It is, therefore, sometimes said that

the construction with `is` *enforces*
a single expansion of the type.

EXAMPLE

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

The system replaces the expression `Endomorphism of H` by the expression `Homomorphism of H,H`.

WARNINGS

1. It is incorrect to redefine (see *25. Redefinitions*) types defined by `is`. This can easily be explained by reference to the example given above. Now the expression `Homomorphism of H,H` is not a pattern of a mode because the argument `H` occurs in that expression twice. By redefining the mode `Endomorphism of H` we would in fact redefine the mode `Homomorphism of H,H`, and thus we would redefine an expression which is not a pattern of a mode.

2. It is, moreover, incorrect to redefine by the construction `is`.

22. Attributes and clusters

22.1. Attributes

The definition of *attribute* has the following form:

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

where Δ is the symbol of attribute and should be entered in the vocabulary and preceded by the classifier `V`.

The types θ_1 , ... , θ_n are called *the parameters of the attribute* Δ.

The expression "$-> \theta(x_1$, ... , $x_n)$" in the definition of the attribute Δ indicates that that attribute may be added to any type that expands to the type $\theta(x_1$, ... , $x_n)$.

Let $\Theta(\Delta)$ be the set of all types expanding to the type $\theta(x_1$, ... , $x_n)$. The type $\theta(x_1$, ... , $x_n)$ will be called *the mother type of the attribute* Δ. Thus $\Theta(\Delta)$ is the set of all types expanding to the mother type of the attribute Δ.

Let τ be a term, and $\vartheta(\tau)$, the type of the term τ. The expression

$$\tau \texttt{ is } \Delta$$

where $\vartheta(\tau) \in \Theta(\Delta)$, will be called *attributive formulas*. The following equivalence holds:

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

EXAMPLE 1

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

The type `Lattice` is the mother type of the attribute `distributive`. Let

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

be the set of all types that expand to the mother type `Lattice`. The attribute `distributive` is a function defined on the set of those terms whose types expand to the mother type of the attribute distributive and have values in the set of the attributive formulas A:

$$\text{distributive: } \{\tau : \vartheta(\tau) \in \Theta(\text{distributive})\} -> \text{A}.$$

EXAMPLE 2

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

The type `set` is the mother type of the attribute `non-empty`. This example is drawn from the article `HIDDEN`.

EXAMPLE 3

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

The type `set` is the mother type of the attribute `finite`.

22.2. Registrations

Attributes are used to construct types. This will be explained by the example below.

EXAMPLE 4

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

where θ is a type such that $\theta \in \Theta$ (non-empty).

In particular, when taking `typ` θ `= Set-Family`, which expands to the type `set`, that is $\theta \in \Theta$ (non-empty), we obtain the type `non-empty Set-Family`.

The condition of existence in the above definition states that there is an object `x` of the type θ such that `x is non-empty`, which is to say that `x` $\neq \emptyset$.

Such definitions will be termed *registrations*.

Of course, it may sometimes be convenient to introduce an abbreviation for the registered attribute. On the basis of the above definition we may introduce the abbreviations of the types `non-empty` θ for $\theta \in \Theta$ (non-empty):

EXAMPLE 5

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

In particular, for θ = `set` and θ = `Subset of X` we obtain the following abbreviations:

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

and

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

The types `DOMAIN` and `SUBDOMAIN` have been introduced in the article `HIDDEN`.

Let us consider an example of an attribute with a parameter.

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

At present there is temporary restriction pertaining to the use of attributes with a parameter: they may be defined, but it is not allowed to register types which includes attributes with a parameter.

22.3. Clusters

A *cluster* is a set $\{\alpha_1 , \ldots , \alpha_n\}$, where $\alpha_1 , \ldots , \alpha_n$ are attributes. Clusters are used in the construction of types.

The registration in Example 4 is the registration of a cluster with one attribute. And here is an example of the registration of a cluster with two attributes.

EXAMPLE 6

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

where θ is a type such that $\theta \in \Theta$ (non-empty) $\cap \Theta$ (finite).

The above registration has the same effect as the registration with a changes order of attributes:

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

The present example is intended to emphasize that the order of attributes in a cluster has no significance whatever for the system.

22.4. Application of attributes

The attributes are use for:

a) construction of attributed types:

non-empty θ,
distributive Λ,
non-empty finite θ,
distributive complemented Λ,

where $\theta \in \Theta$(non-empty), $\Lambda \in \Theta$(distributive).

In particular, the attributed types are

non-empty set,
distributive Lattice,
non-empty finite set,
distributive complemented Lattice.

For each of these attributed types one can register some abbreviation. The first two of the above attributed types will be abbreviated into `DOMAIN` and `D_Lattice`, respectively.

b) formation of attributive formulas:

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

where τ, λ are terms such that

$\vartheta(\tau) \in \Theta$(non-empty), $\vartheta(\lambda) \in \Theta$(distributive).

22.5. Attributive formulas versus qualifying formulas

Note that a *qualifying formula* is any formula of the form ξ `is` θ, where ξ is any term and θ is any type.

The Reader is warned against confusing the attributive formulas (see point *b* above), which are formulas of a new kind, with the *qualifying formulas* formed by attributed types (see point *a* above):

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

where $\theta \in \Theta$ (non-empty), $\Lambda \in \Theta$ (distributive), and ξ is any term.

In particular the *qualifying formulas* are:

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

where ξ is any term.

Formulas of both kinds are three-place ones, the first place being a term and the second place being is. The differences between these two kinds are as follows:

- The type of the term in an attributive formula must expand into the mother type of the attribute.
  The term in the *qualifying formula* is arbitrary.

- The third place in an attributive formula is an attribute.
  The third place in a *qualifying formula* is a type.

22.6. Order of attributes

Generally speaking, the *attributed type* has the form:

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

where $\alpha_1 \ , \ \ldots \ , \ \alpha_n$ are attributes, while θ is a type without an attribute.
The attributes $\alpha_1 \ , \ \ldots \ , \ \alpha_n$ will be termed attributes of the type θ', and the type θ, the core of the type θ'.
Obviously, the use of the type θ' is allowed only after the defining of the clusters of the attributes $\alpha_1 \ , \ \ldots \ , \ \alpha_n$.

At this point it must once more be emphasized that the order of the attributes in the attributed type is of no significance for the system.

The general form of the attributive formula is

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

where α_1 `,` $\ldots$ `,` α_n are attributes, and

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

The content of this formula is the conjunction

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

From the non-commutativity of conjunction in the Mizar language it follows accordingly that the order of the attributes in an attributive formula is essential.

22.7. Principle of the inheritance of registrations

Let K = $\{\alpha_1 , \ldots , \alpha_n\}$ be a cluster, and P = $\{\alpha_{i_1} , \ldots , \alpha_{i_n}\}$ be any subcluster, so that P $\subseteq$ K. The registration of the cluster K makes the registration of the subcluster P superfluous. Thus after having registered the type finite non-empty Set-Family we need register neither the type finite Set-Family nor the type non-empty Set-Family.

The principle of the inheritance of registration is very convenient as it simplifies the introduction of attributed types.

23. Definition of predicate

Here is the most frequent definition of predicate:

```
definition let x1 be θ1 , ... , xn be θn ;
 pred xi1 , ... ,xik Π xj1 , ... ,xjl means
 :L: δ(x1 , ... ,xn );
end;
```

where Π is the symbol of predicate and should be entered in the vocabulary and preceded with the classifier R.

In the definitions of predicates no condition of correctness is required. And here is the form of the definitional theorem:

```
        for x1 being θ1 , ... , xn being θn holds
xi1 , ... , xik Π xj1 , ... , xjl iff δ(x1 , ... , xn );
```

EXAMPLE
The definition of a public predicate given below

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

has the following corresponding definitional theorem:

```
    f is_one_to_one iff
        for x1,x2 st x1 ∈ dom f & x2 ∈ dom f &
              f.x1 = f.x2 holds x1 = x2;
```

24. Definition of functor

Here is the most frequent form of a definition of a public functor:

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

where φ is the symbol of a functor and should be entered in the vocabulary and preceded by the classifier `0`.

In the above definition of public functor two conditions of correctness are required: that of `existence` and that of `uniqueness`. Now let δ`(`x_1`, ... ,`x_n`,x)` be the definiens occurring in a simple definition of a public functor.

The condition of existence should include the justification of the following theorem:

`ex x being` θ`(`x_1`, ... ,` x_n`) st` δ`(`x_1`, ... ,` x_n`,x);`

EXAMPLE

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

The condition of uniqueness should include the justification of the following theorem:

`for y,z being` θ`(`x_1`, ... ,` x_n`) st` δ`(`x_1 `, ... ,` x_n`,y) &`
δ`(`x_1`, ... ,` x_n`,z) holds y = z;`

EXAMPLE

```
definition let f,Y;
  func f"Y -> set means
  :INVERSE_IMAGE: for x holds x ∈ it iff x ∈ dom f & f.x ∈ Y;
  existence from Separation;
  uniqueness
   proof
    let X1,X2 such that
     A1:  x ∈ X1 iff x ∈ dom f & f.x ∈ Y and
     A2:  x ∈ X2 iff x ∈ dom f & f.x ∈ Y;
    x ∈ X1 iff x ∈ X2;
     proof x ∈ X1 iff x ∈ dom f & f.x ∈ Y by A1;
      hence thesis by A2;
     end;
    hence X1 = X2 by TARSKI:2;
   end;
  end;
```

And here is the form of a definitional theorem for public functors:

`for` x_1 `being` θ_1`, ... ,` x_n `being` θ_n`, y being` θ`(`x_1`, ... ,` x_n`) holds`
`y =` `(`x_{i1}`, ... ,` x_{ik}`)` φ`(`x_{j1}`, ... ,` x_{jl}`) iff` δ`(`x_1`, ... ,` x_n`,y);`

EXAMPLE

Let us consider the following definition:

```
definition let f;
 func dom f -> set means
```

```
 :DOM: for x holds x ∈ it iff ex y st [x,y] ∈ graph f;
 existence ... ;
 uniqueness ... ;
end;
```

It has the corresponding definitional theorem:

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

25. Redefinitions

Redefinitions are an important kind of definitions. A redefinition in most cases consists in a change of a specification, that is in the narrowing down of the original mother type. Sometimes it consists in a change of a definiens into an equivalent one. In the former case we justification that the new mother type expands to the original one, and in the latter, that the new and the original definiens are equivalent to one another. These are conditions of the correctness of a redefinition: `correctness` in the case of a change of a specification, and `compatibility`, in the case of a change of a definiens.

From the syntactic point of view redefinitions are a kind of definitions, the only discriminant of a redefinition being the reserved word `redefine`, which should be placed before the word `mode`, `pred`, or `func`.

We shall confine ourselves here to the discussion of redefinitions by a change of a specification.

Let it be said once more: the admissible change of a specification consists in the *narrowing down of the type*. In other words, the new type, that is the one introduced in the redefinition, is a *narrowing down of the original type*, given in the original definition.

The principle of the *narrowing down of the type* yields the following recommendations pertaining to the introduction of concepts:

definitions should be as general as possible and then they should be gradually narrowed down.

The practice of writing Mizar articles has proved so far that the non-observance of that principle results in essential complications.

In the change of a specification we usually omit the definiens. As has been said above, in such a case the original definiens is the definiens understood.

Here is the most frequent form of the redefinition of a mode:

```
definition let x_1 be θ_1, ... , x_n be θ_n;
 redefine mode ? of x_i1, ... ,x_ik -> θ(x_1, ... ,x_n);
 coherence justification ;
end;
```

In the justification of this redefinition we prove the following sentence:

```
for x being ? of x_i1, ... , x_ik holds x is θ(x_1, ... , x_n);
```

And here comes the typical redefinition of a public functor:

```
definition let x_1 be θ_1, ... , x_n be θ_n;
 redefine func (x_i1, ... ,x_ik)φ(x_j1, ... ,x_jl) -> θ(x_1, ... ,x_n);
 coherence justification ;
end;
```

In the justification we prove:

$$(x_{i1}\text{ , ... , }x_{ik})\ \varphi\ (x_{j1}\text{ , ... , }x_{jl})\ \texttt{is}\ \theta(x_1\text{ , ... , }x_n)\ .$$

EXAMPLE 1

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

Note that in the pattern of the functor `g·f` the variable `X` does not occur. This is permissible because the variable `X` is "reproducible" by means of the variable `f` or the variable g.

Data Base

26. Content of the Data Base

The *Data Base* of the Mizar system is organized in the directory \MIZAR. It consists of two subdirectories:

\DICT
\PREL

The subdirectory \DICT contains vocabulary files with the extensions .voc. They are prepared by the author of a given article (or else he avails himself of the already existing ones to suit his needs).

The files with the extensions .voc contain declarations of symbols of modes, predicates, functors, attributes, structures, selectors, and function brackets to be used in a given article. The declaration of a symbol consists of the classifier of that symbol itself. Moreover the files contain indications of the priorities of functors. Priority in that case means the information about the strength of binding.

The subdirectory \PREL contains files with the extensions:

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

They are formed by the LIBRARIAN from a given article and contain definitions (.nfr, .def, .dno, .dco), theorems (.the), and schemes (.sch), adequately processed and encoded. The appropriate files from that directory are drawn by the Mizar ACCOMMODATOR if in the declaration of the environment there are directives joining definitions, theorems, and schemes in the article to which one refers.

Some users organize in the Mizar directory their own subdirectories, such as

\MML
\ABSTR

They form **Main Mizar Library**, play the role of documents and are not used by the system.

The \MML subdirectory includes files with the extensions .miz, which contain Mizar articles.

The subdirectory \ABSTR contains files with extensions .abs. They are abstracts of articles and are obtained from the original text of a given article by the elimination of lemmas, definitions of private objects, and all proofs, and by the current numbering of theorems and definitions. The user should refer to those numbers of theorems and definitions if he needs them in his own article (the numbering in the original file is of local importance, only within the text of a given article).

27. Directives

In the environment we may place directives of five kinds, namely:

– *vocabulary directive*

vocabulary α_1 , ... ,α_n ;

– *signature directive*

signature β_1 , ... ,β_n ;

– *definition directive*

definitions β_1 , ... ,β_n ;

– *theorem directive*

`theorems` β_1 `,` ... `,`β_n`;`

– *schema directive*

`schemes` β_1 `,` ... `,`β_n`;`

– *cluster directive*

`clusters` β_1 `,` ... `,`β_n`;`

where α_1 `,` ... `,`α_n are names of vocabularies, and β_1 `,` ... `,`β_n are names of articles.

In the course of processing the system observes the order of directives. That may be essential with reference to the signature directives, which contain information about the definitions (including redefinitions) of the same symbol. As has been said, one and the same symbol may be defined many times, and the binding definition is always that of the last signature specified in the environment. If the order is incorrect, then earlier directives may be overridden by later ones.

EXAMPLE

The symbol `U` is a symbol of a functor. We defined `A U B` twice:

1. in the article α`.miz`
   - as the `DOMAIN` which is a sum of domains, and
2. in the article β`.miz`
   - as the `set` which is a sum of sets.

As is known, the type `DOMAIN` expands to the type `set`.

Suppose that in the article γ_1`.miz` and γ_2`.miz` we have adopted the directives `signature` α`,` β`;` and `signature` β`;` , respectively. Let `A`, `B` be domains.

The term `A U B` in γ_1`.miz` is of the type `set`. Why is that so? The PROCESSOR tries to adjust the types of the variables `A`, `B` to the definition 2. It obviously succeeds in that because the types of the variables `A`, `B` expand to the type `set`.

The term `A U B` in γ_2`.miz` is of the type `DOMAIN`, because the processor can apply to that term the definition 1.

NOTE: The term

```
A U B exactly DOMAIN
```

has in γ_1`.miz` the type `DOMAIN`.

39. Name of article

The file with an article must have the extension `.miz` and is termed Mizar article. The name of a Mizar article consists of at most eight signs: letters, figures, connectors `_` and apostrophe `'`. The name of a Mizar article may be neither a numeral nor a reserved word.

REMARK: The usage is that the letters occurring in the name of a file should be capital letters. Hence `TARSKI` is agreement with that usage while `tarski` and `Tarski` are not.

29. Vocabulary

The vocabulary lists the symbols together with their qualifiers. It also indicates the binding strength of the functor symbols.

29.1. Name of the vocabulary

The vocabulary forms the file α`.voc`
where α is the *name-of-vocabulary-file*. The name α may, but need not, be a name of a Mizar article. For instance, the name of the file `FUNC.voc` is not derived from a Mizar article: in the Data Base there are articles `FUNCT_1.miz`, `FUNCT_2.miz`, ... , but - at least for the time being - there is no article `FUNC.MIZ`.

29.2 Qualifiers

The symbols `M`,`R`,`O`,`G`,`U`,`K`,`L`,`V` called *qualifiers*, are used to introduce symbols, respectively:

`M` *mode,*
`R` *predicate,*
`O` *functor,*
`G` *structure,*
`U` *selector,*
`K` *right functor bracket,*
`L` *left functor bracket,*
`V` *attribute.*

EXAMPLE
Below there is the vocabulary file `FUNC.voc`:

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

The first column houses qualifiers, and those from the second to the space house symbols. After the space we may indicate the binding strength, also called priority. This applies only to the symbols of functors. The binding strength of a given functor is indicated by the number which follows its symbols.

Now `graph`, `id`, `.` are functor symbols. `Function` is a mode symbol, and `is_one_to_one` is a predicate symbol. The numbers `128` and `100` denote the magnitude of priority. Priority is indicated only in the case of functor symbols; thus the functor symbols `graph` and `id` have the priority `128`, and the functor symbol `.` has the priority `100`.

REMARK

1. The binding strength of predicate symbols is not indicated because these always bind more weakly than functor symbols do.

2. The number indicating the priority of a given functor must be separated from the symbol of that functor by at least a single space.

3. There may not be even a single space between a qualifier and the corresponding vocabulary symbol.

29.3. Strength of binding

The greater the number indicating the binding strength the greater the binding strength of the functor symbol concerned. The priority of a functor is a natural number ranging from 0 to 255. In the case of some functor symbol the number indicating its priority may be missing. That means that that symbol has the standard priority. The standard binding strength is 64.

The concept of the binding strength of functor symbols is linked to the sequence of the operation with a given binding strength. The memorization of the priorities of at least some symbols may be largely used for the elimination of superfluous brackets.

Part Two: The System

30. Preliminary information on the system

The Mizar computer system processes and verifies formalized theorems in the Mizar language, or, to be more precise, the Mizar article which contains those theorems. That is achieved by two programs of the PC Mizar system, called ACCOMMODATOR and PROCESSOR. The ACCOMODATOR draws from Data Base the necessary information on the basis of the directive to be found in the environment of the article, and the PROCESSOR verifies the text of the article.

Mizar articles are usually written by means of editors integrated with the system. The most popular ones are multiEdit, or so-called Mem, and Brief.

Suppose that we write an article `GEOM.miz`. We successively call the programs ACCOMMODATOR and PROCESSOR by

`accom geom` and `mizar geom`,

respectively.

After the termination of the processing of each program the listing will show the inscription `THANKS OK` or `SORRY`, according to whether the article is correct or contains errors. The numbers of the errors are indicating in the text, which makes their correction easier.

But it may also occur that the processing is interrupted. One of the reasons of the interruption of the processing is the bad installation of the system or discrepancy between the private Data Base and the public Data Base. So-called bugs, or errors in the system, are another, very rare, cause. If the system does not accept your proof and you think that it should you are advised to consult the authors of the implementation of the system PC Mizar, namely A. Trybulec or Cz. Byliński.

A continuous verification of the article being written is recommended. That makes it possible to eliminate errors currently and facilitates formalization. Working without notes, that is with the computer, is recommended, too. Of course, major problems may be solved more conveniently on a separate sheet of paper or on a blackboard.

31. Programs of the PC Mizar system

The PC Mizar system cover Data Base and a number of programs, of which three are essential, i.e., closely linked to the system:

`ACCOM.exe` (ACCOMMODATOR),
`MIZAR.exe` (PROCESSOR), and
`EXTRACT.exe` (EXTRACTOR).

To simplify the description we assume that all files are placed in standard directories.

32. Installation of the PC Mizar system

The PC Mizar system is implemented on computers compatible with IBM PC XT/AT; Mizar articles can be written on such computers only.

The PC Mizar system may be installed on hard disc on the directory `C:\MIZAR`. That directory includes the basic programs of the system, system files, and input macrocommands. The files of the Data Base of the Main Mizar Library are also placed in that directory.

Note that <*name-of-article*> is the name of the file containing a Mizar text. The text should have the extension to `.miz`, e.g., `BOOLE.miz`. That extension is always implicitly

joined by the ACCOMMODATOR, PROCESSOR, and EXTRACTOR to the name of the text. The name of the text is written without extension.

33. Accommodator

The ACCOMMODATOR is called:

`accom` <*name-of-article*>

We shall now briefly describe the basic program `ACCOM.exe`. The ACCOMMODATOR draws from Data Base the required items of information on the basis of the directives written out in the environment of the article concerned. The items of information thus received are recorded by the ACCOMMODATOR on the following files:

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

The information contained in the above files covers, among other things, a summary vocabulary and descriptions of formats, signatures, definienses, statements, and schemes.

Next to that the ACCOMMODATOR forms the file <*name-of-article*>`.err`, which record the errors in the environment. In that file the successive lines have triplets of numbers which in turn indicate respectively the number of the line, the number of the column, and the number of the error. Indentifying the errors on the basis of that file is, as can be seen, not very convenient, and this is why we suggest another method.

Following the command:

`errrp` <*my-directory*>`\`<*name-of-article*>`.miz c:\mizar\mizar`

the program forms the file <*name-of-article*>`.lst`, so-called listing, with errors entered on the copy of the text of the article in question. Further the file with the explanation of errors, `MIZAR.msg`, is joined following command

`c:\mizar\mizar`

When using the ACCOMMODATOR one has always to bear in mind the fact that the PROCESSOR communicates with the library solely through the files prepared by the ACCOMMODATOR. Hence changes in the environment and/or the Data Base require a new accommodation.

By restricting the calling of the ACCOMMODATOR to possible changes in the environment and/or the Data Base we considerably speed up the processing of the article. Obviously, in order to avail ourselves of those advantages we must see to it that the ACCOMMODATOR and the PROCESSOR be installed independently of one another and called by two different commands.

34. Processor

The calling of the processor of PC Mizar (and next of the so-called ERRPRINTER, which forms the so-called listing - both terms will be explained below) has the following form:

`miz` <*name-of-article*>

When the text is being processed we can, at any moment, interrupt the functioning of the PROCESSOR by pressing the keys **Ctrl Break**. This interrupts the analysis of the text by the PROCESSOR, an error as the signal of that interruption (error `#1255`) being indicated in the line in which the interruption took place.

The PROCESSOR yields the list of the errors discovered during the processing of the text. The list of the errors is placed in the file bearing the same name as the file with the input text and the extension `.err`. It is formed in the same directory as the Mizar text. If the text is correct the file remains empty. The file with the errors includes the list of the descriptions of those errors in the form of triplets of numbers. The description of a given error includes its position in the text (i.e., the number of the line and that of the column) and the number of the error. The file of errors may be used for various purposes:

1. through the editors (e.g., Mem) or other programs, for finding and identifying their position in the text and their identification by the number;

2. for forming a listing by the so-called ERRPRINTER (program `ERRP`).

The ERRPRINTER reads the text and the file with the errors and produces the listing. The listing thus formed is a document which includes a copy of the text with the indication of the errors (in the listing) and joined explanation of the errors found in the text. If the text is correct, then the listing, instead of the explanations of errors, is concluded by the inscription `Thanks OK` . The listing has the same name as the input text but with the extension `.lst`. The listing is formed in the same directory as the text being analysed.

35. Extractor

The EXTRACTOR is called by:

`extract` <*name-of-article*>

The EXTRACTOR forms the following files:

<*name-of-article*>`.nfr` - formats,
<*name-of-article*>`.dno` - patterns of modes, predicates, and functors, and types of the arguments occurring in the patterns,
<*name-of-article*>`.def` - definienda,
<*name-of-article*>`.dco` - description of the types of the results,
<*name-of-article*>`.sch` - list of schemes and their description,
<*name-of-article*>`.the` - list of theorems and their description,
<*name-of-article*>`.dcl` - list of clusters.

These files should be transferred to the directory `\PREL`.

36. Preparation of abstracts

If a Mizar article is correct, then it can be turned into an abstract using the program `MIZ2ABS`, called ABSTRACTOR (see *26. Content of the Data Base*) on the directory `\ABSTR`. The program also forms the file of the errors found in the course of processing.

The ABSTRACTOR turns a Mizar article into its abstract. The abstract has the same name as the article plus the extension `.abs`. That file is formed in the same directory as the file with the Mizar article in question. The abstract is practically a copy of that part of the article which contains definitions of public objects and theorems. Lemmas, proofs, and definitions of private objects are disregarded.

37. Use of a correct articles

A verified article may be passed to the Main Mizar Library. Moreover, it may be included in the local library with which one works. This facilitates, and in most cases is indispensable for, the writing of the next article: usually it is so that our articles are thematically interconnected, not to say that they are continuations of earlier ones.

For that purpose we have:

1. To form the files of the Data Base.
2. To transfer of the files of the Data Base to the local data base.

Moreover it is worth while to prepare an abstract.

38. Practical advice for users

1. Start writing an article from the basic definitions and the basic theorems (i.e., *statements* and *schemes*).
2. Form the vocabulary and place in it the symbols with their qualifiers. In the case of operations possibly include in the vocabulary their respective priorities.
3. Write out the directives.
4. Add the reservations.
5. Precede definitions with theorems which may be useful in justifying the conditions of the correctness of those definitions.
6. Write out full skeletons of the proofs, but without `by` and `from`, i.e., without the so-called simple justifications. If the errors marked `#4` are the only ones, then the skeleton are correct. Fight the errors until only the "fours" are left.
7. Add auxiliary theorems so as to shorten the proofs by eliminating recurrent modules of reasonings. In Mizar it pays because Mizar proofs are fully formalized.
8. It is worth while introducing a number of private objects - variables, predicates, functors - in order to shorten the proofs.
9. Finally add simple justification and start eliminating the "fours". The main source of errors is making use of two universal sentences after one `by`. Unfortunately, proofs must sometimes be lengthened - do not expect too much of Mizar. A stronger CHECKER could be programmed but at the cost of a longer waiting time.

Afterword

We have disregarded here the following subjects which, we think, are less important for the beginners in using Mizar:

- proving *per cases*,
- proving by definitional expansion,
- compound definitions,
- definitions with assumption,
- definitions of private functors and predicates,
- understood properties,
- antonyms and synonyms.

The author presents this *Outline of PC Mizar* to the Readers with mixed feelings. The text seems immature and probably requires various modifications. But I have been told that even such a text may prove useful in the use of Mizar. I hope that the reading of this *Outline* will do more good than harm. The Readers are warmly requested to send all their comments, which are likely to be important in my further work on this *Outline*, to the address:

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

Bibliography

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

[2] Rasiowa, H., Introduction to Modern Mathematics, Amsterdam – London – Warsaw 1973.

[3] Pogorzelski, W.A., Słupecki, J., O dowodzie matematycznym (Mathematicals Proofs), Warsaw 1970.

[4] Pogorzelski, W.A., Klasyczny rachunek kwantyfikatorów (The Classical Functional Calculus), Warsaw 1970.

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

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

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