# Chapter 1

# Writing a Mizar article in nine easy steps

After reading this chapter and doing the exercises, you will be able to write a basic Mizar article all by yourself.

The chapter presents an explanation of the nine steps needed to finish a Mizar article. For each step it also shows how this turns out for a specific example. There are exercises too, so you can test your understanding.

To be able to read this chapter you should know some basic mathematics but not much. You should know a little bit about:

- first order predicate logic
- some basic set theory

Mizar is based on set theory on top of predicate logic, that's why.

The most difficult part of writing Mizar is finding what you need in the MML which is the name of the *Mizar Mathematical Library.* Unfortunately that problem has no easy solution. You will discover that apart from that, writing a Mizar article is straight-forward.

## 1.1 Step 1: the mathematics

Before you start writing a Mizar article you need to decide what it should be about. It's very important not to start writing the article too soon! A proof that takes five minutes to explain – after which people look into the air and then mumble: 'hm, yes, I see' – takes about a week to formalize. So every improvement to the mathematics that you can make *before* you start formalizing will pay off very much.

**The example** As a running example in this chapter we write a small Mizar article called `my_mizar.miz`. It is about *Pythagorean triples.* Those are triples of natural numbers $\{a, b, c\}$ for which holds that:

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

The Pythagorean theorem tells us that these triples correspond to square angled triangles that have sides of integer length. The best known of these triples are $\{3, 4, 5\}$ and $\{5, 12, 13\}$ but there are an infinite number of them.

The theorem that we will formalize says that all Pythagorean triples are given by the formula:

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

or are multiples of such a triple. For instance the triple $\{3, 4, 5\}$ is given by $n = 2$, $m = 1$, and the triple $\{5, 12, 13\}$ is given by $n = 3$, $m = 2$.

The proof of the theorem is straight-forward. You only needs to consider the case where $a$ and $b$ are relative prime. Some thought about parity gives that one of the $a$ and $b$ has to be even, and that the other and $c$ is odd. Then if $b$ is the even number of the two we get that:

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

But if the product of two relative prime numbers is a square, then both of those numbers are squares. So you get:

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

which leads to the required formula.

**Exercise 1.1.1** Study the Mizar Mathematical Library and try to decide whether Mizar is equally suited to all fields of mathematics and computer science, or that Mizar is best suited to certain subjects. In the latter case, what is most easily formalized in Mizar? And what least?

**Exercise 1.1.2** We mentioned the *Pythagorean theorem* which says that the sides $a$, $b$ and $c$ of a square angled triangle satisfy $a^2 + b^2 = c^2$. Try to find this theorem in the MML. If it's there, in what article does it occur and what is the theorem's reference? If not, or if you don't like the version that you find (exercise for when you finished this chapter): write an article that proves it.

## 1.2 Step 2: the empty Mizar article

In order to write a Mizar article you need a working Mizar system and a file to put the Mizar article in. In fact, as you will find out, you will need *two* files: a `.miz` file for your article and a `.voc` file for its vocabulary.

In this tutorial we follow the Windows conventions of naming files. For Mizar under Unix the backslashes should go the other way. So what is called `text\my_mizar.miz` here, under Unix should be `text/my_mizar.miz`.

We will assume that you already have a properly installed Mizar system. For a description of how to install Mizar see the `readme.txt` file that is in the Mizar distribution (under Unix it is called `README`).

To write your article, you need to be in a directory that has two subdirectories called `text` and `dict`. If those directories don't exist yet, make them. Put in the `text` directory an empty file called `my_mizar.miz` and put in the `dict` directory an empty file called `my_mizar.voc`. (Replace the `my_mizar` with a more appropriate name if you are writing an article of your own.)

The smallest legal `.voc` file is empty but the smallest legal `.miz` file looks like this:

```
environ
begin
```

Use your favorite text editor to put those two lines in the `my_mizar.miz` file. Then check it using the command:

```
mizf text\my_mizar.miz
```

It will both check syntax and mathematics of your article. If everything is well this command prints something like:

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

Verifier, Mizar Ver. 7.0.01 (Win32/FPC)
Copyright (c) 1990,2004 Association of Mizar Users
Processing: text\my_mizar.miz

Parser   [    2]   0:00
Analyzer   0:00
Checker  [    1]   0:00
Time of mizaring: 0:00
```

meaning that this 'article' contains no errors. (If you use the `emacs` editor with its Mizar mode you don't need to type the `mizf` command. In that case all you need to do to check the file is hit `C-c RET`.)

Apart from the `my_mizar.miz` file, the `text` directory now contains 25 other files. You never need to look inside them. They are only used internally by the Mizar system.

The next step is to connect the `.miz` file to the `.voc` vocabulary. Add a `vocabularies` directive to your article:

```
environ
 vocabularies MY_MIZAR;

begin
```

(Important: the name of the vocabulary has to be written in capitals! This is a remnant of Mizar's DOS origins.)

Now the article is ready to be written. For the moment add some line of text at the end of the file just to find out what happens if the article contains errors:

```
environ
 vocabularies MY_MIZAR;

begin
 hello Mizar!
```

After you run the `mizf text\my_mizar.miz` command again, Mizar will insert the error messages *inside* your file. It will put a `*` with a number below any error. Also there will be an explanation of those error numbers at the end of the file. So after running the `mizf` command again your file looks like:

```
environ
 vocabularies MY_MIZAR;

begin
 hello Mizar!
::>  *143,321

::> 143: No implicit qualification
::> 321: Predicate symbol or "is" expected
```

There is no need to try to understand those two error messages, because of course `hello Mizar!` is not legal Mizar. So remove this line, now that you have seen what Mizar thinks of it. You don't need to remove the error message lines. They vanish the next time you run `mizf`.

The lines containing error messages can be recognized because they start with `::>`. Mizar only gives you error numbers. It never prints anything about why it thinks it is wrong. Sometimes this lack of explanation is frustrating but generally Mizar errors are quite obvious.

**Exercise 1.2.1** The minimal Mizar article that we showed is 2 lines long. What is the shortest article in the MML? What is the longest? What is the average number of lines in the articles in the MML?

As a rule of thumb an article less than a 1000 lines is considered too short to be submitted to the MML. However several articles in the MML are less than a 1000 lines long since articles are shortened during revision of the MML. How many of the articles in the MML are currently shorter than a 1000 lines?

**Exercise 1.2.2** Copy an article from the MML to your private `text` directory. Check that the Mizar checker `mizf` processes it without error messages. Experiment with making small modifications to it and see whether Mizar can detect where you tampered with it.

Put one screen-full of non-Mizar text – for instance from a Pascal or C program – somewhere in the middle of the article. How many error messages will you get? Can Mizar sufficiently recover from those errors to check the second half of the file reasonably well?

## 1.3 Step 3: the statement

To start translating your mathematics to Mizar you need to write the theorem that you want to prove in Mizar syntax.

There are two things about Mizar syntax that are important for you to note:

- There are no spelling variants in Mizar. Although Mizar resembles natural language a lot, it is a formal language and there are no possibilities to choose between phrasings. For example:

  `and` means something different from `&`

  `not` means something different from `non`

  `such that` means something different from `st`

  `assume` means something different from `suppose`

  `NAT` means something different from `Nat` means something different from `natural`

  So you should really pay attention to the exact keywords in the Mizar syntax. It's not enough if it resembles it.

  The only exception to this rule is that `be` and `being` are alternative spellings. So although it's more natural to write `let X be set` and `for X being set holds` ... you are also allowed to write `let X being set` and `for X be set holds` ...

- There is no distinction in Mizar between 'function notation' and 'operator notation'. In most programming languages something like `f(x,y)` and `x+y` are syntactically different things. In Mizar this distinction doesn't exist. In Mizar *everything* is an operator. If you write `f(x,y)` in Mizar then it really is a 'operator symbol' `f` with zero left arguments and two right arguments.

  Similarly predicate names and type names can be any string of characters that you might wish. You can mix letters, digits and any other character you might like in them. So for instance if you want to call a predicate `\/-distributive` you can do so in Mizar. And it will be one 'symbol'.

  If you are not sure what characters go together as a symbol in a Mizar formula, you can go to the web pages of the MML abstracts. In those pages the symbols are hyperlinks that point to definitions. So just click on them to find out what symbol they are.

To write Mizar you need to know how to write terms and how to write formulas.

We tackle this in top-down fashion. First we describe how to translate formulas from predicate logic into Mizar. Then we describe how the syntax of Mizar terms works.

### 1.3.1 Formulas

Here is a table that shows all you want to know about how to write predicate logic in Mizar:

| | |
|---|---|
| $\perp$ | `contradiction` |
| $\neg\phi$ | `not` $\phi$ |
| $\phi \wedge \psi$ | $\phi$ `&` $\psi$ |
| $\phi \vee \psi$ | $\phi$ `or` $\psi$ |
| $\phi \Rightarrow \psi$ | $\phi$ `implies` $\psi$ |
| $\phi \Leftrightarrow \psi$ | $\phi$ `iff` $\psi$ |
| $\exists\, x.\, \psi$ | `ex x st` $\psi$ |
| $\forall\, x.\, \psi$ | `for x holds` $\psi$ |
| $\forall\, x.\, (\phi \Rightarrow \psi)$ | `for x st` $\phi$ `holds` $\psi$ |

There is no special way to write $\top$ in Mizar. One usually writes `not contradiction` for this. Note that with the quantifiers it should be `st` and not `such that`. Also note that `for x st` $\phi$ `holds` $\psi$ is just 'syntactic sugar' for `for x holds (`$\phi$ `implies` $\psi$`)`. After the parser processed it, the rest of the system will not see any difference between those two formulas.

Using this table you now can write logical formulas in Mizar.

There is one more thing that you need to know to write Mizar formulas in practice. Mizar is a *typed* language. We will discuss Mizar types in detail in Section 1.3.4 below, but the types turn up in the predicate logic formulas too. All variables that you use in formulas need to have a type. There are two ways to give a variable a type:

- Put the type in the formula using a `being` type attribution. For instance you can write an existential formula about natural numbers $m$ and $n$ as:

  ```
  ex m,n being Nat st ...
  ```

- Give the type for the variable with a `reserve` statement. This doesn't introduce a variable, it just introduces a notation convention. If you use a `reserve` statement you can leave out the type in the formula:

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

  This way of typing variables in formulas is much more convenient than explicitely typing the variables and so it is the method that is generally used.

**Exercise 1.3.1** Translate the following Mizar formulas into predicate logic notation:

$\phi$ `iff not not` $\phi$

`not (`$\phi$ `&` $\psi$`) implies not` $\phi$ `or not` $\psi$

`ex x st` $\phi$ `implies` $\psi$

`for x st for y st` $\phi$ `holds` $\psi$ `holds` $\chi$

To do this exercise you need to know the priorities of the logical operators in Mizar. They are the usual ones in predicate logic:

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

This means that you should read `not` $\phi$ `implies` $\psi$ as `(not` $\phi$`) implies` $\psi$ (because the `not` binds stronger), but that `ex x st` $\phi$ `implies` $\psi$ means `ex x st (`$\phi$ `implies` $\psi$`)`.

**Exercise 1.3.2** Translate the following predicate logic formulas into Mizar syntax:

$$\neg\phi \Leftrightarrow \neg\neg\neg\phi$$

$$\neg(\phi \vee \psi) \Rightarrow (\neg\phi \wedge \neg\psi)$$

$$\exists\, x.\, (\phi \wedge \psi)$$

$$\exists\, x.\, \big((\exists\, y.\, (\phi \Rightarrow \psi)) \Rightarrow \chi\big)$$

### 1.3.2 Relations

You still need to know how to write atomic formulas. In Mizar there are two ways to write those:

- If $R$ is a *predicate* symbol, write:

  $$x_1\,,\, x_2\,,\, \ldots,\, x_m\ R\ x_{m+1}\,,\, \ldots,\, x_{m+n}$$

  Both $m$ or $n$ might be 0, in which case this becomes prefix or postfix notation. Note that postfix notation can have more than one argument, as for instance in `x,y are_relative_prime`. Please note that brackets around the arguments of the predicate are not allowed.

- If $\mathcal{T}$ is a *type*, or the adjectives part of a type, write:

  $x$ `is` $\mathcal{T}$

  For instance you can write `x is prime`. We will discuss types in Section 1.3.4 below.

To find out what relations are available to you, you will need to browse the MML. Here is a short list of some of the most frequent used relations to get you started.

| | |
|---|---|
| $=$ | `=` |
| $\neq$ | `<>` |
| $<$ | `<` |
| $\leq$ | `<=` |
| $\in$ | `in` |
| $\subseteq$ | `c=` |

The first character in the Mizar representation of $\subseteq$ is the letter `c`.

### 1.3.3 Terms

There are three ways to write Mizar terms:

- If $f$ is an operator symbol, which Mizar calls a *functor*, write:

  $$(x_1, x_2, \ldots, x_m)\ f\ (x_{m+1}, \ldots, x_{m+n})$$

  Again, $m$ or $n$ might be 0, in which case this becomes prefix or postfix notation. As an example of a postfix operator there is `^2` for square. The brackets around either list of arguments can be omitted if there is just one argument.

- If $\mathcal{L}$ and $\mathcal{R}$ are *bracket* symbols, write:

  $$\mathcal{L}\ x_1, x_2, \ldots, x_n\ \mathcal{R}$$

  In this case brackets `(` and `)` around the arguments are not necessary, since the symbols themselves are already brackets. An example of this kind of term is the ordered pair `[x,y]`. In that case $n = 2$, $\mathcal{L}$ is `[` and $\mathcal{R}$ is `]`.

- Any natural number is a Mizar term. If you write natural numbers, you should add to your `environ` the line:

  ```
  requirements SUBSET, NUMERALS, ARITHM;
  ```

  because else they won't behave like natural numbers.

The word *functor* for function symbol or operator symbol is Mizar terminology. It has nothing to do with the notion of functor in category theory. It is used to distinguish a function symbol in the logic from a function object in the set theory (which is a set of pairs).

Again to find out what operators are available to you, you will need to browse the MML. Here is a short list of some of the most frequent used operators to get you started.

| | |
|---|---|
| $\emptyset$ | `{}` |
| $\{x\}$ | `{x}` |
| $\{x,y\}$ | `{x,y}` |
| $X \cup Y$ | `X \/ Y` |
| $X \cap Y$ | `X /\ Y` |
| $\mathbb{N}$ | `NAT` |
| $\mathbb{Z}$ | `INT` |
| $\mathbb{R}$ | `REAL` |
| $x + y$ | `x + y` |
| $x - y$ | `x - y` |
| $-x$ | `-x` |
| $xy$ | `x*y` |
| $x/y$ | `x/y` |
| $x^2$ | `x^2` |
| $x^y$ | `x to_power y` |
| $\sqrt{x}$ | `sqrt x` |
| $\mathcal{P}(X)$ | `bool X` |
| $(x,y)$ | `[x,y]` |
| $X \times Y$ | `[:X,Y:]` |
| $Y^X$ | `Funcs(X,Y)` |
| $f(x)$ | `f.x` |
| $\langle\rangle$ | `<*>D` |
| $\langle x\rangle$ | `<*x*>` |
| $\langle x,y\rangle$ | `<*x,y*>` |
| $p \cdot q$ | `p^q` |

The digit `2` is part of the symbol for the square operator. The last four operators are used to build finite sequences over the set $D$.

The `.` operation is function application *inside the set theory*. If `f` is a symbol *in the language* representing a functor with zero left arguments and one right argument then you write:

```
f x
```

or (you can always put a term in brackets):

```
f(x)
```

If `f` is a function in the set theory – a set of pairs – then

```
f.x
```

is the image of `x` under `f`, in other words it is the `y` such that `[x,y] in f`.

**Exercise 1.3.3** Translate the following Mizar formulas into mathematical notation:

```
NAT c= REAL

1/0 = 0
```

```
sqrt -1 in REAL

sqrt(x^2) <> x

{x} /\ {-x} = {x} /\ {0}

[x,y] = {{x,y},{x}}

p = <*p.1,p.2*>
```

**Exercise 1.3.4** Translate the following mathematical formulas into Mizar syntax:

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

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

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

$Y^X \in \mathcal{P}(\mathcal{P}(X \times Y))$

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

$p \cdot \langle \rangle = p$

$f(g(x)) \neq g(f(x))$

### 1.3.4 Types: modes and attributes

The one thing left for you to understand to write Mizar formulas is Mizar's types. Although Mizar is based on ZF-style set theory – so the *objects* of Mizar are untyped – the *terms* of Mizar are typed.

An example of a Mizar type is:

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

This type should be parsed as:

$$\boxed{\boxed{\texttt{non empty}}\ \boxed{\texttt{finite}}}\ \boxed{\underline{\texttt{Subset}}\ \texttt{of NAT}}$$

A Mizar type consists of an instance of a *mode* with in front a cluster of *adjectives*. The type without the adjectives is called the *radix type*. In this case the mode is `Subset` and its argument is `NAT`, the radix type is `Subset of NAT`, and the two adjectives are `non empty` and `finite`.

To put this abstractly, a Mizar type is written:

$$\alpha_1\ \ \alpha_2\ \ \ldots\ \ \alpha_m\ \ \mathcal{M}\ \texttt{of}\ x_1, x_2, \ldots, x_n$$

where $\alpha_1, \ldots\ \alpha_m$ are adjectives, $\mathcal{M}$ is a mode symbol and $x_1, x_2, \ldots, x_n$ are terms. The keyword `of` binds the arguments of the mode to the mode. It's like the brackets in a term.

The number of arguments $n$ is part of the definition of the mode. For instance for `set` it is zero. You can't write `set of` ..., because there does not exist a `set` mode that takes arguments.

Modes are dependent types. To find out what modes are available to you, you will need to browse the MML. Here is a short list of some of the most frequent used modes to get you started.

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

Note that there both are modes `Relation` and `Function` with no arguments, and more specific modes `Relation of X,Y` and `Function of X,Y` with two arguments. They share the mode symbol but they are different.

Also note that the modes depend on *terms*. So there are no types representing the functions from a type to a type. The `Function` mode represents the functions from a set to a set. As an example the function space $(X \to Y) \times X \to Y$ corresponds to the Mizar type `Function of [:Funcs(X,Y),X:],Y`

Adjectives restrict a radix type to a subtype. They either are an *attribute*, or the negation of an attribute using the keyword `non`. Again, to find out what attributes are available to you, you will need to browse the MML. Here is a list of a few attributes.

`empty`
`even`
`odd`
`prime`
`natural`
`integer`
`real`
`finite`
`infinite`
`countable`

Maybe you now wonder about how types are used in Mizar. To clarify this take a look at three Mizar notions – `NAT`, `Nat` and `natural` – that all mean the same thing: the set of natural numbers. Here is a table that compares their uses:

| *meaning* | *declaration* | *formula* |
|---|---|---|
| $n \in \mathbb{N}$ | | `n in NAT` |
| $n : \mathbb{N}$ | `n be Nat` | `n is Nat` |
| $n : \mathbb{N}$ | `n be natural number` | `n is natural` |

`NAT` is a term, `Nat` is a type and `natural` is an adjective. `be`/`being` are a typing in a declaration and go between a variable and a type. `in` is the $\in$ relation of the set theory and goes between two terms. `is` goes between a term and a type or between a term and an adjective. Note that in Mizar you can have a 'type judgement' as a formula in the logic.

**The example** We now give the statement of the theorem for the example. The statement that we will prove in this chapter is:

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

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

So put this after the `begin` line of your `my_mizar.miz` file.

We also could have made the quantification of the `a`, `b` and `c` explicit:

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

but it is not necessary. Mizar automatically quantifies over free variables.

We now analyze this statement in some detail. The formula has the structure:

$$\phi_1 \ \texttt{\&}\ \phi_2 \ \texttt{\&}\ \phi_3 \ \texttt{implies ex m,n st}\ \phi_4 \ \texttt{\&}\ \phi_5 \ \texttt{\&}\ \phi_6 \ \texttt{\&}\ \phi_7\texttt{;}$$

The predicate logic formula that corresponds to this is:

$$(\phi_1 \wedge \phi_2 \wedge \phi_3) \Rightarrow \exists\, m, n.\, (\phi_4 \wedge \phi_5 \wedge \phi_6 \wedge \phi_7)$$

The first three atomic formulas in this are:

$$\begin{aligned} \phi_1 &\equiv \texttt{a\^{}2 + b\^{}2 = c\^{}2} \\ \phi_2 &\equiv \texttt{a,b are\_relative\_prime} \\ \phi_3 &\equiv \texttt{a is odd} \end{aligned}$$

They have the structure:

$$\begin{aligned} \phi_1 &\equiv t_1 \boxed{\texttt{=}}\, t_2 \\ \phi_2 &\equiv t_3\,,\, t_4 \boxed{\texttt{are\_relative\_prime}} \\ \phi_3 &\equiv t_5\ \texttt{is}\ \mathcal{T} \end{aligned}$$

In this `=` and `are_relative_prime` are predicate symbols. $\mathcal{T}$ is the adjective `odd`. So we here see both kinds of atomic formula: twice a relation between terms and once a typing.

The first term $t_1$ in $\phi_1$ is:

$$t_1 \equiv \texttt{a\^{}2 + b\^{}2}$$

It has the structure:

$$\begin{aligned} t_1 &\equiv u_1 \boxed{\texttt{+}}\, u_2 \\ u_1 &\equiv v_1 \boxed{\texttt{\^{}2}} \\ u_2 &\equiv v_2 \boxed{\texttt{\^{}2}} \\ v_1 &\equiv \texttt{a} \\ v_2 &\equiv \texttt{b} \end{aligned}$$

In this `+` and `^2` are functor symbols.

**Exercise 1.3.5** Find Mizar types for the following concepts:

odd number that is not a prime number

empty finite sequence of natural numbers

uncountable set of reals

element of the empty set

non-empty subset of the empty set

What is the problem with the last two concepts? Do you think they should be allowed in Mizar? Study this manual to find out whether they are.

**Exercise 1.3.6** Write the following statements as Mizar formulas:

The only even prime number is 2.

If a prime number divides a product it divides one of the factors.

There is no biggest prime number.

There are infinitely many prime twins.

Every even number $\geq 4$ is the sum of two primes.

Write these statements first using `reserve` statements. Then write them again but this time with the types in the formulas.

## 1.4 Step 4: getting the environment right

Add the statement that you wrote to your article. Then check it. You will get error messages:

```
environ
 vocabularies MY_MIZAR;

begin
 reserve a,b,c,m,n for Nat;
::>                      *151

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

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

That's because you didn't import what you used from the MML. There's nothing wrong with the statement, there's something wrong with the environment (it's empty). To correct this, you need to add *directives* to the `environ` part of the article to import what you need.

It is hard to get the environment of a Mizar article correct. In practice people often just copy the environment of an old article. However that doesn't help much when it doesn't work, and occasionally it doesn't work. So you still will have to understand how the environment works, to get it right when you get environment related errors.

**Exercise 1.4.1** Try to use an environment of an existing article from the MML. Do the errors go away? If so, you might consider the rest of *Step 4* for now, and just use this environment.

### 1.4.1 Vocabulary, notations, constructors

The rule is quite simple. For everything you use – predicate, functor, mode or attribute – you have to add a relevant reference to three directives:

- `vocabularies`
- `notations`
- `constructors`

The list of references for `notations` and `constructors` is generally almost identical. In fact, if you follow the algorithm from this section to get them right they *will* be identical. These directives are about:

- Lexical analysis. The tokens in a Mizar article come from lists called *vocabularies*. Mizar variables are identifiers with a fixed syntax, but the predicates, functors, types and attributes all are *symbols* which can contain any character. You need to indicate from what vocabularies you use symbols.

- Parsing of expressions. To have an expression you need to list the articles that you use predicates, functors, types and attributes from. The `notations` directive is for the syntax of expressions. The `constructors` directive is for its meaning.

Here is a list of what was use in the statement, what kind of thing it is, and what vocabularies and articles you need for it:

| *symbol* | *kind* | *vocabulary* | *article* |
|---|---|---|---|
| `=` | *pred* | | `HIDDEN` |
| `<=` | *pred* | `HIDDEN` | `XXREAL_0` |
| `+` | *func* | `HIDDEN` | `XCMPLX_0` |
| `*` | *func* | `HIDDEN` | `XCMPLX_0` |
| `-` | *func* | `ARYTM_1` | `XCMPLX_0` |
| `Nat` | *mode* | `HIDDEN` | `NAT_1` |
| `are_relative_prime` | *pred* | `ARYTM_3` | `INT_2` |
| `^2` | *func* | `SQUARE_1` | `SQUARE_1` |
| `odd` | *attr* | `MATRIX_2` | `ABIAN` |

You don't need to refer to the `HIDDEN` vocabulary or the `HIDDEN` article but you need to list the others. The vocabularies should go in the `vocabularies` directive and the articles should go both in the `notations` and `constructors` directives. So your environment needs to be:

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

Here is the way to find out what the vocabulary and article a given symbol are:

- To find the vocabulary use the command:

  `findvoc -w '`*symbol*`'`

  For instance the command:

  ```
  findvoc -w '-'
  ```

  gives:

```
FindVoc, Mizar Ver. 7.0.01 (Win32/FPC)
Copyright (c) 1990,2003 Association of Mizar Users
vocabulary: ARYTM_1
O-  32
```

The O means that this is a functor symbol, and 32 is the priority.

- To find the article, the easiest way is to go to the web pages of the abstracts of the MML on the Mizar web site, find a use of the symbol somewhere, and click on it to go to the definition.

**Exercise 1.4.2** Find out what is in the HIDDEN vocabulary by typing:

```
listvoc HIDDEN
```

For each of the 25 symbols in this vocabulary, find an article that introduces a notation that uses it.

### 1.4.2 Redefinitions and clusters

But now things get tough. It turns out that your environment still does not work! The problem is that the types are wrong. If you check the article with the new environment you get:

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

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

To get rid of this kind of typing problems is the most difficult part of writing a Mizar article. (In practice this kind of error often is caused because a *cluster* is missing. So if you are completely mystified by a Mizar typing error, start thinking 'cluster!')

The errors that you see here are indeed almost all caused by the fact that some clusters are not imported in the environment. This causes the expressions not to have the right types. For instance the first error is caused by the fact that `a^2` and `b^2` have type `Element of REAL`, while the `+` wants its arguments to have type `complex number`. We will show in detail why this is the case, and how to fix this by importing the right clusters.

A Mizar expression doesn't have just one type, it has a whole *set of types*. For instance, with the right environment the number `2` has the following types:

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

There are two ways to influence the types of an expression.

- To give a specific expression a more precise radix type, you use *redefinitions*. A functor can have many redefinitions that cause it to get different types depending on the types of its arguments.

  Here is an example of a redefinition taken from `PEPIN`:

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

  What this does is change the type of some of the terms that use the `^2` operator.

  The original definition (of which this is a redefinition) is in `SQUARE_1`:

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

  In `SQUARE_1` there already are two other redefinitions of `^2`:

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

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

  Now suppose that in your environment you have the directive:

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

Then if you write an expression $t$`^2`, the type of this expression will depend on the type of $t$. All definitions and redefinitions of `^2` will be considered in the order that they are imported through the `notations` directive. So in this case the definition for `complex number` is first, then there is the redefinition for `Element of COMPLEX`, then the redefinition for `Element of REAL`, and finally (go to the next article, which is `PEPIN`) there is the redefinition for `Nat`.

Now the rule is that the *last* redefinition that fits the type of all arguments applies.

So if $t$ has type `Nat` then $t$`^2` has type `Nat` too, while if $t$ does not have any of the types `Element of COMPLEX` or `Element of REAL` or `Nat`, then it will have type `set`.

Note that since the the order of the articles in the `notations` directive is important for this!

- To generate more adjectives for an expression, Mizar has something called *clusters*. The process of adding adjectives according to these clusters is called the *rounding up* of clusters.

  Here are three examples of clusters, taken from `FINSET_1`.

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

  This means that every `set` that has adjective `empty` also will get adjective `finite`.

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

  This means that every expression that has type `Subset of B` where `B` has type `finite set` also will get adjective `finite`.

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

  This means that every expression of the shape `X \/ Y` where `X` and `Y` have type `finite set` also will get adjective `finite`.

  These examples show both kinds of cluster that add adjectives to the set of types of an expression. The first two do this based on the *type* of the expression (this is called 'rounding up' a type), and the third add adjectives based on the *shape* of the expression.

To summarize: redefinitions are for narrowing the radix type and clusters are for extending the set of adjectives. (There are also redefinitions that have nothing to do with typing – because they redefine something different from the type – and a third kind of cluster that has nothing to do with adding adjectives. You should not be confused by this.)

You can always test whether some type $\mathcal{T}$ is in the set of types of an expression $t$ by changing it to `(`$t$ `qua` $\mathcal{T}$`)`. You get an error if $t$ didn't have $\mathcal{T}$ in its set of types. In that case you might start looking for a redefinition or cluster that changes this situation.

**The example** The first error in the example:

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

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

is that the `+` in `a^2 + b^2` is not well-typed. (The `*` of the error message `*103` indicates the position of the error. In this case it is below the `+` symbol.) The definition of `^2` that is in effect here is from `SQUARE_1`:

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

and the definition of `+` is from `XCMPLX_0`:

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

So this shows that you would like `a^2` and `b^2` which have type `Element of REAL` to also get type `complex number`. This means that you want these expressions to get an extra adjective – the adjective `complex` – and so you need a cluster. (Once again: for more adjectives you need clusters, while for a more precise radix type you need a redefinition.)

It turns out that an appropriate cluster is in `XCMPLX_0`:

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

After you add

```
registrations XCMPLX_0;
```

to the environment the first error indeed is gone and you only have two errors left:

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

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

The next error (the `*102` below the `<=`) is similar to the previous one. The `<=` predicate expects arguments of type `real number`, but `m` and `n` have type `Nat`. If you study the MML for some time you will find that `Nat` is the same as `Element of omega` (the definition of `Nat` in `NAT_1` and the synonym in `NUMBERS`.)

Therefore the following two clusters give you what you need. From `ARYTM_3`:

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

and from `XREAL_0`:

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

The cluster directive now has become:

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

With this directive in the environment the only error left is:

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

::> 103: Unknown functor
```

This error is caused by the fact that Mizar does not consider `2` to be a `Nat`. You will see below, after the discussion of the `requirement` directive, how to get rid of this final error.

### 1.4.3 The other directives

Here is a list of the eight kinds of directives in the `environ` and the kind of reference they take:

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

Here is when you need the last four kinds of directive:

- The `definitions` directive is *not* about being able to use the theorems that are part of the definitions (that's part of the `theorems` directive.) It's about automatically unfolding predicates in the thesis that you are proving.

  This directive is useful but not important. You can ignore it until you get up to speed with Mizar.

- The `theorems` and `schemes` directives list the articles you use theorems and schemes from. So whenever you refer to a theorem in a proof, you should check whether the article is in the list of this directive.

  These are easy directives to get right.

- The `requirements` directive makes Mizar know appropriate things automatically.

  For instance to give numerals type `Nat` you need

  ```
  requirements SUBSET, NUMERALS;
  ```

  This is the solution to the last typing error left in your article.

  With `ARITHM` Mizar also knows some basic equations about numbers automatically. For instance with it, Mizar accepts `1+1 = 2` without proof.

  This is an easy directive to get right. Just put in all requirements and be done with it.

So now that you got the environment right, the article checks like this:

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

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

a^2 + b^2 = c^2 & a,b are_relative_prime & a is odd implies
```

```
 ex m,n st m <= n & a = n^2 - m^2 & b = 2*m*n & c = n^2 + m^2;
::>                                                          *4

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

The only error left is *4. It means that Mizar is not able to prove the statement on its own.

This is an important milestone. Articles with only *4 errors are 'almost finished'. They just need a bit of proof.

**Exercise 1.4.3** For each of the following statements find an environment that makes all errors different from error *4 go away:

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

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

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

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

**Exercise 1.4.4** Consider the nine types:

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

There are 81 pairs of different types $\mathcal{T}_1$ and $\mathcal{T}_2$ from this list for which the formula:

for x being $\mathcal{T}_1$ holds (x qua $\mathcal{T}_2$) = x;

should be allowed (however not all of them are provable in the MML). What is an environment that gives the minimum number of error messages `*116` (meaning `Invalid "qua"`)? What is used from the articles in this environment?

**Exercise 1.4.5** Apart from the kinds of cluster that we showed in Section 1.4.2 (the ones that generate extra adjectives for terms), Mizar has something called *existential clusters*. These are the clusters without `->`. They don't generate extra adjectives for a term. Instead they are needed to be allowed to add adjectives to a type.

The reason for this is that Mizar types always have to be non-empty. So to be allowed to use a type something has to be proved. That is what an existential cluster does.

For example to be allowed to use the type:

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

you needs an existential cluster from `GROUP_2`:

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

If you don't have the right existential clusters in your article you will get error `*136` which means `non registered cluster`.

Which types in the following list have an existential cluster in the MML? For the ones that have one, where did you find them? For the ones that don't have one, what would be an appropriate existential cluster?

```
empty set

odd prime Nat

infinite Subset of REAL

non empty Relation of NAT,NAT
```

## 1.5 Step 5, maybe: definitions and lemmas

Step 5 you can skip. Then you just start working on the proof of the theorem immediately.

However if you did step 1 right, you probably have some lemmas that you know you will need. Or maybe you need to define some notions before being able to state your theorem at all.

So do so in this step. Add relevant definitions and lemmas to your article now. Just like the statement in step 3. Without proof. Yet.

### 1.5.1 Functors, predicates

There are two ways to define a functor in Mizar:

- As an abbreviation:

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

  The `coherence` correctness condition is that the expression `x * x` really has type `complex number` like it was claimed. If Mizar can't figure this out by itself you will have to prove it.

- By a characterization of the result:

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

  In the characterizing statement the variable `it` is the result of the functor.

  The `existence` and `uniqueness` are again correctness conditions. They state that there always exists a value that satisfies the defining property, and that this value is unique. (In the proof of them you are allowed to use that the assumption from the definition is satisfied.)

  In this case `existence` is the statement:

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

  and `uniqueness` is the statement:

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

And here is an example of the way to define a predicate:

- 
```
definition let m,n be Nat;
 pred m,n are_relative_prime means
  m hcf n = 1;
end;
```

  The definition of a predicate doesn't have a correctness condition. (In this example `hcf` is the greatest common divisor.)

### 1.5.2 Modes and attributes

There are two ways to define a mode in Mizar:

- As an abbreviation:

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

  In this definition `quasi_total` and `Function-like` are attributes.

- By a characterization of its elements.

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

  The `existence` correctness condition states that the mode is inhabited. This has to be the case because Mizar's logic wants all Mizar types to be non-empty.

  In this case `existence` is the statement:

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

And here is an example of the way to define an attribute:

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

**The example** The essential lemma for the example is that if two numbers are relative prime and their product is a square, then they are squares themselves. In Mizar syntax this is the statement:

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

Now the attribute `square` is already present in the MML, but it is in the article `PYTHTRIP`, and that is the article that you are currently writing! So let's prentend it is not there and add it to the article, to practice writing definitions.

The way to define the attribute `square` (this will enable you to have a type `square number`) is:
```

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

(Note that we do not only define the attribute `square` for expressions of type `Nat`, but for all Mizar terms. Else statements like `not -1 is square` would not be well typed.)

Part of most definitions is the introduction of a new symbol. In this case it is the attribute symbol `square`. Again, it is already in the MML, in vocabulary `PYTHTRIP`. But again, we do not want to use that, so let's add it to the vocabulary of the `MY_MIZAR` that you are writing.

So add the line:

```
Vsquare
```

to the `my_mizar.voc` file.

The `V` in front means that this is an attribute symbol. Vocabularies have an `O` in front for functors, `R` in front for predicates, `M` in front for modes and `V` in front for attributes.

The statements in a Mizar article can be local to the article or be visible to the outside of the article. In the latter case they have to be preceded by the keyword `theorem`. So add `theorem` in front of the statements, like this:

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

theorem Th2:
 a^2 + b^2 = c^2 & a,b are_relative_prime & a is odd implies
  ex m,n st m <= n & a = n^2 - m^2 & b = 2*m*n & c = n^2 + m^2;
::>                                                           *4

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

(Note that there are *two* `*4` errors below theorem `Th1`. The reason for this is that the Mizar checker considers the two conclusions `m is square` and `n is square` to be two different proof obligations.)

**Exercise 1.5.1** Show that in Mizar definition by abbreviation is secondary to definition by characterization:

How can you simulate the `equals` variant of a `func` definition with the `means` variant of a `func` definition? Similarly: how can you simulate the `is` variant of a `mode` definition with the `means` variant of a `mode` definition?

**Exercise 1.5.2** We claimed that Mizar's types are always non-empty. However you can write the type:

```
Element of {}
```

Explain why this is not a problem.

## 1.6 Step 6: proof skeleton

Mizar is a block structured language like the Pascal programming language. Instead of `begin/end` blocks it has `proof/end` blocks. And instead of procedures it has theorems. Apart from that it is a rather similar language.

We will now write the proof of the theorem. For the moment we just write the steps and not link them together. Linking the steps in the proof will be step 7 of the nine easy steps.

A Mizar proof begins with the keyword `proof` and ends with the keyword `end`.

Important! If you have a proof after a statement there should not be a semicolon after the statement. So:

*statement* $\boxed{;}$
`proof`
  *proof steps*
`end;`

is wrong! It should be:

*statement*
`proof`
  *proof steps*
`end;`

This is easy to do wrong. Then you will get a `*4` error at the semicolon that shouldn't be there.

In this section we will discuss the following kinds of proof step. There are some more but these are the frequently used ones:

| *step* | *section* |
|---|---|
| *compact* | 1.6.2 |
| `assume` | 1.6.1 |
| `thus` | 1.6.1 |
| `let` | 1.6.1 |
| `take` | 1.6.1 |
| `consider` | 1.6.2 |
| `per cases/suppose` | 1.6.2 |
| `set` | 1.6.3 |
| `reconsider` | 1.6.3 |
| *iterative equality* | 1.6.4 |

### 1.6.1 Skeleton steps

During the steps of the proof the statement that needs to be proved is kept track of. This is the 'goal' of the proof. It is called the *thesis* and can be referred to with the keyword `thesis`.

A Mizar proof consists of steps. Steps contain statements that you know to be true in the given context. Some of the steps reduce the thesis. At the `end` of the proof the thesis should be reduced to $\top$ or else you will get error `*70` meaning `Something remains to be proved`. Steps that change the thesis are called *skeleton steps*.

The four basic skeleton steps are `assume`, `thus`, `let` and `take`. They correspond to the shape of the thesis in the following way:

| *thesis to be proved, before* | *the step* | *thesis to be proved, after* |
|---|---|---|
| $\phi$ `implies` $\psi$ | `assume` $\phi$ | $\psi$ |
| $\phi$ `&` $\psi$ | `thus` $\phi$ | $\psi$ |
| `for x being` $\mathcal{T}$ `holds` $\psi$ | `let x be` $\mathcal{T}$ | $\psi$ |
| `ex x being` $\mathcal{T}$ `st` $\psi$ | `take` $t$ | $\psi[\mathtt{x} := t]$ |

Just like with `for` the typing can be left out of the `let` if the variable is in a `reserve` statement.

Here is an example of how these skeleton steps work in a very simple proof:

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

At the start of the proof the thesis is `for x,y st x = y holds y = x`. After the `let` step it is reduced to `x = y implies y = x`. After the `assume` step it is reduced to `y = x`. After the `thus` step it is reduced to $\top$ and the proof is complete.

The skeleton steps of the proof of the example will be:

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

At this point we don't have anything to put in the place of the $m$ and $n$. We need another kind of step for that.

**Exercise 1.6.1** Explain why you can always end a proof with a `thus thesis` step. This is the Mizar version of QUOD ERAT DEMONSTRANDUM. Count in the MML how many times this construction occurs. (Include in your count `hence thesis` which is the same thing.) Is there an occurence of `thesis` in the MML which is *not* part of this construction?

**Exercise 1.6.2** Write the skeleton steps in the proofs for the following Mizar statements:

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

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

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

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

Write a fresh variable in the `take` steps if there is no good choice for the term.

### 1.6.2 Compact statements and elimination

The simplest step is a *compact step*. This is just a statement that is true in the current context. This is the most common step in a Mizar proof. So a Mizar proof skeleton for the most part just looks like:

...
*statement*;
*statement*;
*statement*;
...

where each statement in the list is a consequence of a combination of some of the preceding statements.

And then there are the `consider` and `per cases`/`suppose` constructions. They are both related to a statement:

| *if you can prove this* | ... | *you can have this as a proof step* |
|---|---|---|
| `ex x st` $\phi$ | | `consider x such that` $\phi$; |
| $\phi_1$ `or` $\phi_2$ `or ... or` $\phi_n$ | | `per cases;`<br>`suppose` $\phi_1$;<br>*proof for the case* $\phi_1$<br>`end;`<br>`suppose` $\phi_2$;<br>*proof for the case* $\phi_2$<br>`end;`<br>...<br>`suppose` $\phi_n$;<br>*proof for the case* $\phi_n$<br>`end;` |

To use natural deduction terminology, these steps correspond to existential and disjuction *elimination*. The `consider` step introduces a variable that you can refer to in your proof, just like the `let` step.

We can now add the following steps to the example proof to give us terms `m` and `n` for the `take` step:

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

The first four steps are compact statements. The two `consider` steps use the existential statement that's part of the definition of `square`.

### 1.6.3 Macros and casts

Often in proofs certain expressions occur many times. It is possible to give such expressions a name using `set` commands:

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

A `set` command is a definition of a constant that is local to the proof. It behaves very much like a *macro*.

When writing Mizar articles, often you want to use an expression with a different type than it's got. You can change the type of an expression with the `reconsider` command. This is like a `set`, but also gives the type of the new variable.

In the example we will need the abbreviated variables `h`, `m2` and `n2` to have type `Nat`. This can be accomplished by changing the `set` lines in the previous paragraph to:

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

Again the new variables behave like macros, but now with a different type. So `reconsider` is a way to *cast* the type of a term.

### 1.6.4 Iterative equalities

In mathematical calculations one often has a chain of equalities. Mizar has this feature too.

You can write a calculation:

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

in Mizar as:

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

Such a chain of `.=` equalities is called an *iterative equality.* Note that the first equality in the chain is written with the `=` symbol instead of with the `.=` symbol.

**Exercise 1.6.3** Collect the steps for the proof of `Th2` that were given in this section. Put them in the proper order in your `my_mizar.miz` file. Use the `reconsider` lines (not the `set` lines) and replace the abbreviated expressions everywhere by their abbreviation. Put in the iterated equality as well.

Add two compact statements before the `reconsider` lines stating that `b` is even and that `c` is odd. They will be needed to justify the `reconsider` lines. Add two compact statements relating `m2` and `n2` to `a` and `c` after the `reconsider` lines.

Now check your file with `mizf`. Update the environment if necessary until you have only `*4` errors left. How many `*4` errors do you get?

## 1.7 Step 7: completing the proof

In this section you will add justifications to the steps of your proof. That will finish your article.

### 1.7.1 Getting rid of the *4 errors

There are three ways to justify a step in Mizar:

- By putting a semicolon `;` after it:

  *statement* `;`

  This is the empty justification. It tells Mizar: *figure this out by yourself.*

- By putting `by` after it with a list of labels of previous statements:

  *statement* `by` *reference* , ... , *reference* ;

  In fact the previous way to justify a step is the special case of this in which the list of references is empty.

- By putting a subproof after it:

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

  The subproofs are what gives Mizar proofs the block structured look.

  If you don't use `thesis` in a subproof, the statement can be reconstructed from the skeleton steps in that subproof. Therefore you might want to omit the statement. You can do this by using `now`:

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

  This is exactly the same as the statement with the subproof, but you don't need to write the statement.

Here is a small example that shows how `by` is used:

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

Of course this one is so easy that Mizar can do it on its own:

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

Often in the list of references after a `by` justification, there is a reference to the previous statement. This can also be written by putting the keyword '`then`' in front of the step. Using `then` you can often avoid having to label statements. So you can replace:

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

by

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

Some people like writing the `then` on the previous line, after the statement that it refers to:

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

This is a matter of taste. Other people choose the position of the `then` depending on whether the statement after the `then` has a label or not.

When you want to use `then` to justify a `thus` step you are not allowed to write `then thus`. You have to write `hence`. In Mizar when in certain combinations of two keyword get together you have to replace the combination by something else. Here are the two relevant equations:

$$\begin{array}{rcccl} \texttt{then} & + & \texttt{thus} & = & \texttt{hence} \\ \texttt{thus} & + & \texttt{now} & = & \texttt{hereby} \end{array}$$

So the `hence` keyword means two things:

1. This follows from the previous step.
2. This is part of what was to be proved.

A common Mizar idiom is `hence thesis`. You see that many times in any Mizar article.

### 1.7.2 Properties and requirements

So now you need to hunt the MML for relevant theorems. This is made difficult by three reasons:

- The MML is big.
- The MML is not ordered systematically but chronologically. It is like a mediaeval library in which the books are put on the shelf in the order that they have been acquired.
- Mizar is smart. The theorem you can use might not look very much like the theorem you are looking for.

As an example of the fact that theorems can be in unexpected places, the basic theorem that relates the two kinds of dividability that are in the Mizar libary:

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

is in an article called `SCPINVAR` which is about loop invariants of a certain small computer program (SCP stands for SCMPDS program and SCMPDS stands for small computer model with push-down stack).

The best way to find theorems in the MML is by using the `grep` command to search all the `.abs` abstract files. Although the MML is huge it's small enough for this to be practical. For instance this is an appropriate command to search for the theorem in the `SCPINVAR` article:

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

Another problem when looking for theorems is that Mizar is too smart.

If you look at the definitions of `+`, `*` and `<=` you will find that they contain the keywords `commutativity`, `reflexivity`, `connectedness`, `synonym` and `antonym`:

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

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

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

These keywords mean two things:

- If you use a `synonym` or `antonym` then Mizar will internally use the real name. So if you write:

  ```
  a < b
  ```

  then internally Mizar will consider this to be an abbreviation of:

  ```
  not b <= a
  ```

  and if you write:

  ```
  a >= b
  ```

  then internally Mizar will consider this to mean:

  ```
  b <= a
  ```

- Mizar will always 'know' when justifying a step, about the properties `commutativity`, `reflexivity` and `connectedness`. So if you have a statement:

  ```
  ... x*y ...
  ```

  then Mizar behaves as if this statement is exactly the same as:

  ```
  ... y*x ...
  ```

  Also Mizar will use all implication of the shape:

  $$x = y \Rightarrow x \leq y$$

  and:

  $$x \leq y \vee y \leq x$$

  when trying to do a justification. These properties mean for the $<$ relation that:

  $$\neg(x < x)$$

  and:

  $$x < y \Rightarrow x \leq y$$

  For instance if you have proved:

  ```
  a < b
  ```

  then you can refer to this when you only need:

  ```
  a <> b
  ```

This is all very nice but let's look at three examples of the subtlety it leads to:

- Suppose you want to justify the following step:

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

  It turns out that the theorem that you need is:

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

  It will give you:

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

  but because of the `antonym` definition of `<` and `>` this really means:

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

which is equivalent to:

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

Which is what you need.

- Suppose you need to prove the equation:

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

  The `+` and `-` operators have the same priority and are left associative, so this has to be read as:

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

  Because of `commutativity` of `+` this really is the same as:

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

  So you need to show that: `c + (c + a) = (c + c) + a` which is associativity of `+`, a theorem from the `XCMPLX_1` article:

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

  Note that the original equation doesn't look very much like an instance of associativity!

- Consider the transitive law for `<=`:

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

  It's good to have this, but where is the analogous law for `<`:

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

  It turns out that this also is `AXIOMS:22`! To see why this is the case, note that it is a consequence of the stronger theorem:

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

  and becase of the `antonym` definition of `<` this is the same as:

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

which is equivalent to:

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

which is indeed `AXIOMS:22`.

(The 'arithmetical lemmas' for the arithmetical operations that we have used as examples here all are in the articles `AXIOMS`, `REAL_1`, `REAL_2` and `XCMPLX_1`. The rule is that if the lemma that your are looking for is valid in the complex numbers (which generally means that it just involves equality and not `<=` or `<`) then it is in `XCMPLX_1`. Unfortunately if it *does* talk about inequalities, then you will have to look in all the other three articles. In that case there is not an easy rule that will tell you where it has to be.)

Sometimes you should not go look for a theorem. If you have `requirements ARITHM` then Mizar knows many facts about the natural numbers without help. Sometimes those facts even don't have a `theorem` in the library anymore, so you search for a long time if you don't realize that you should use `requirements`. For instance if you use `requirements ARITHM`:

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

and:

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

and:

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

and:

```
0 < 1;
```

and:

```
0 <> 1;
```

all don't need any justification.

Suppose you want to justify:

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

If you had `0 < 1` then this would just be an instance of `AXIOMS:22` (as we just saw). But `requirements ARITHM` gives that to you for free! So you can just justify this step with `AXIOMS:22`.

**Exercise 1.7.1** We claim that the MML is big. Count how many source lines there are in the `.miz` articles in the MML. Try to guess how many lines of Mizar a competent Mizar author writes per day and then estimate the number of man-years that are in the MML.

### 1.7.3 Automation in Mizar

In Mizar it's not possible to write *tactics* to automate part of your proof effort like you can do in other proof checkers. All Mizar's automation is built into the Mizar system by its developers.

Mizar has four kinds of automation:

**Semantic correlates** Internally Mizar stores its formulas in some kind of conjunctive normal form that only uses $\wedge$, $\neg$ and $\forall$.

This means that it automatically identifies some equivalent formulas. These equivalence classes of formulas are called *semantic correlates.* Because of semantic correlates, the skeleton steps of a formula are more powerful than you might expect. For instance you can prove:

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

and also you can prove:

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

(Mizar does not identify $\phi$ `&` $\psi$ with $\psi$ `&` $\phi$. You have to put the `thus` steps in a proof in the same order as the conjuncts appear in the thesis.)

**Properties and requirements** Properties and requirements were discussed in the previous section.

**Clusters** Clusters automatically generate adjectives for expressions. Often theorems can be rephrased as clusters, after which they will become automatic.

Consider for instance the theorem:

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

which says that a square can never be the sum of two odd squares. (The reason for this is that odd squares are always 1 modulo 4, so the sum would be 2 modulo 4, but an even square is always 0 modulo 4.)

This theorem can be rephrased as a cluster:

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

Once you have proved this cluster Mizar will apply the theorem automatically when appropriate and you don't have to give a reference to it in your proofs.

**Justification using `by`** The `by` justifier is a weak first order prover. It tries to deduce the statement in front of the `by` from the statements that are referred to after the `by`.

A `by` justification pretty much feels like a 'natural reasoning step'. When one studies the proofs that humans write, it turns out that they tend to choose slightly smaller steps than `by` can do.

The way `by` works is:

- It puts the implication that it has to prove in conjunctive normal form. Then it tries to do each conjunct separately.

  That's why you often get more than one `*4` erroror for a failed justification. Mizar puts in an error for every conjunct that has not been justified.

- It then tries to prove the inference. In only *one* of the antecedents can a universal quantifier be instantiated. Therefore:

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

  won't work because it both has to instantiate the `for` in `A1` and `A2`. You will need to split this into:

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

- The `by` prover will do congruence closure of all the equalities that it knows. It also will combine the typing information of equal terms.

- A conclusion that has an existential quantifier is equivalent to an antecedent that has a universal quantifier. Therefore `by` is able to derive existential statements from instantiations of it.

### 1.7.4 Unfolding of definitions

Here is a table that shows when Mizar will unfold definitions:

| | |
|---|---|
| func/equals | $-$ |
| func/means | $-$ |
| pred/means | $\diamond$ |
| mode/is | $+$ |
| mode/means | $\diamond$ |
| attr/means | $\diamond$ |
| set | $+$ |

The items in this table that have a + behave like macros. For instance the definition of `Nat` is:

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

If you write `Nat` it's exactly like writing `Element of NAT`. So theorems about the `Element` mode will automatically apply to `Nat`.

The items in the table that have a $-$ will never be expanded. All you get is a definitional theorem about the notion. You will need to refer to this theorem if you want to use the definition.

The items in the table that have a $\diamond$ will be expanded if you use the `definitions` directive. Expansion only happens in the thesis. It takes place if a skeleton step is attempted that disagrees with the shape of the thesis.

For instance the definition of `c=` is:

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

If you have `TARSKI` in your `definitions` you can prove set inclusion as follows:

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

Similarly consider the definition of equality in `XBOOLE_0`:

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

If you add `XBOOLE_0` to your `definitions` it will allow you to prove:

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

or even:

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

**Exercise 1.7.2** Write Mizar proofs of the following three statements:

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

for X holds X c= X;

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

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

that only make use of the following two theorems from the MML:

```
theorem :: TARSKI:2
 for X,Y holds
  X = Y iff for x holds x in X iff x in Y;

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

(If you look in the MML, it will turn out that in the `TARSKI` article this is not exactly what's there. We present it like this for didactic purposes.)

The approach you should follow in your proof, is that in order to prove a statement like `X c= Z` you should first prove that:

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

and then justify the `X c= Z` by referring to this `for` statement, together with `TARSKI:def 3`:

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

Try to make proofs in which the argumentation is as clear as possible, but also make a second set of proofs that are as small as possible.

**Exercise 1.7.3** Take the following skeleton proof of the *the drinker's principle.* It is called the drinker's principle because if `X` is the set of people in a room that are drinking, then it says that *in each room there is a person such that if that person is drinking, then everyone in that room is drinking.* This sounds paradoxical (why should one person have the power to make all other people in the room drink?) but when you really understand what it says, it is not.

```
reserve X,x,y for set;

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

Add labels to the statements and put in justifications for all the steps.

Experiment with other proofs of the same statement. If you know constructive logic: write a Mizar proof of this statement that shows where the non-constructive steps in the proof are.

**Exercise 1.7.4** Write a proof of the following statement:

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

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

Can you think of a proof that doesn't need `per cases`?

**Exercise 1.7.5** Finish the proof of `Th2`. Add as many clusters and lemmas as you need. For instance you might use:

```
Lm1: m is odd square & n is odd square implies
      m + n is non square;
Lm2: n is even iff n/2 is Nat;
Lm3: m,n are_relative_prime iff
      for p being prime Nat holds not (p divides m & p divides n);
Lm4: m^2 = n^2 iff m = n;
```

but if you choose to use different lemmas that's fine too. Just put them before the theorem (you can prove them later if you like). If you don't think a lemma will be useful to others you don't have to put `theorem` in front of it.

Now add compact statements and justifications to the proof of `Th2` until all the `*4` errors are gone.

## 1.8 Step 8: cleaning the proof

Now that you finished your article the fun starts! Mizar has several utilities to help you improve your article. These utilities will point out to you what is in your article that is not necessary.

Those utilities don't put their error messages inside your file on their own. You need to put the name of the program as an argument to the program `revf` to accomplish that.

`relprem` This program points out references in a `by` justifications – as well occurrences of `then` – that are not necessary. These references can be safely removed from your article.

This program also points out the real errors in the article. Some people prefer to always use `relprem` instead of `mizf`.

`relinfer` This program points out steps in the proof that are not necessary. It will indicate references to statements that can be short-circuited. You can omit such a reference and replace it with the references from the step that it referred to.

As an example of `relinfer` consider the following part of a proof:

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

If you run:

```
revf relinfer my_mizar
```

then you will get `*604` meaning `Irrelevant inference`:

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

It means that you can replace the reference to `A2` in step `A4` with the references in the justification of `A2` itself. In this case that's `A1`. So the proof can be shortened to:

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

Please take note that the `relinfer` program is dangerous! Not because your proofs will break but because they might become ugly. `relinfer` encourages you to cut steps that a human might want to see.

`reliters` (pronounced *rel-iters*, not *re-liters*) This program points out steps in an interative equality that can be skipped. If you omit such a step you have to add its references to the references of the next one.

As an example of `reliters` consider the following iterative equality:

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

If you run:

```
revf reliters my_mizar
```

then you will get `*746` meaning `References can be moved to the next step of this iterative equality`:

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

So you can remove the terms with the `*746` from the iterative equality. You then have to move the references in the justification to that of the next term in the sequence. So the iterative equality can be shortened to:

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

`trivdemo` This program points out subproofs that are so simple that you can replace them with a single `by` justification. It's surprising how often this turns out to be the case!

`chklab` This program points out all labels that are not referred to. They can be safely removed from your article.

`inacc` This program points out the parts of the proofs that are not referred to. They can be safely removed from your article.

`irrvoc` This program points out all vocabularies that are not used. They can be safely removed from your article.

`irrths` This program points out the articles in the `theorem` directive that are not used for references in the proofs. They can be safely removed from your article.

Cleaning your article is fun! It really feels like polishing something after it is finished and making it beautiful.

Remember to check your article once more with `mizf` after you have finished optimizing it, to make sure that you haven't accidentally introduced any new errors.

**Exercise 1.8.1** Take the proof that you made in exercise 1.7.3 and run the programs that are discussed in this section on it. Which of the programs advise you to change your proof?

**Exercise 1.8.2** Consider the statements in a Mizar proof to be points in a graph and references to the statements to be the edges. What are the `relprem` and `relinfer` optimizations when you consider them as transformations of these graphs? Draw a picture!

## 1.9 Step 9: submitting to the library

So now your article is finished and clean. You should consider submitting it to the MML.

There are two rules that articles for the MML have to satisfy:

- It is mathematics that's not yet in the MML.

  There should not be alternate formalizations of the same concepts in the MML. There's too much double work in the MML already.

- It is significant.

  As a rule of thumb the article should be at least a 1000 lines long. If it's too short consider extending your article until it's long enough.

There are many reasons to submit your article to the MML:

- People will be able to use your work and build on it.
- Your definitions will be the standard for your subject in later Mizar articles.
- When the Mizar system changes your article will be kept compatible by the people of the Mizar group.
- An automatically generated TeX version of your article will be published in a journal called *Formalized Mathematics*.

To find the details of how to submit an article to the MML go to the web page of the Mizar project and click on the appropriate link. Be aware that to submit your article to the MML you need to sign a form that gives the Mizar people the right to change it as they see fit.

If your article is not suitable for submission to the MML but you still want to use it to write further articles, you can put it in a local library of your own. Currently this kind of local library will only support a few articles before it becomes rather inefficient. The MML has been optimized such that library files don't grow too much. You can't do this optimization for your local library yourself, because the tool for it is not distributed. So it's really better to submit your work to the MML if possible.

If you want a local library of your own there need to be a third directory called `prel` next to the already existing `dict` directory. To put the files for your article in this directory you need to run the command:

```
miz2prel text\my_mizar
```

outside your `text` and `prel` directories (it will put some files in your `prel` directory.) After you have run `miz2prel` you can then use your article in further articles.

**Exercise 1.9.1** How many Mizar authors are there in the MML? Shouldn't you be one?

# Chapter 2

# Some advanced features

Now that you know how to write a Mizar article, let's take a look at some more advanced features of Mizar that you haven't seen yet.

To show these features we use a small proof by Grzegorz Bancerek from the article `quantal1.miz`. This article is about the mathematical notion of *quantales*. It is not important that you know what quantales are. We just use this proof to show some of Mizar's features.

```
reserve
 Q for left-distributive right-distributive complete Lattice-like
     (non empty QuantaleStr),
 a, b, c, d for Element of Q;

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

This proof is about a more abstract kind of mathematics than the example from the previous chapter. Mizar is especially good at formalizing abstract mathematics.

## 2.1 Set comprehension: the Fränkel operator

The proof of theorem `Th5` from `quantal1.miz` contains expressions like:

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

This corresponds to:

$$\{H(a,b) \mid b \in Y\}$$

where $a$ is a fixed parameter and $b$ is allowed to range over the elements of $Q$ that satisfy $b \in Y$.

In Mizar this style of set comprehension is called the *Fränkel operator*. It corresponds to the axiom of replacement in set theory, which was first proposed by Adolf Fränkel, the F in ZF.

The general form of the Fränkel operator is:

`{` *term* `where` *declaration of the variables*`:` *formula* `}`

As you can see in some of the other Fränkel operator terms in the proof, if all variables in the term occur in a `reserve` statement their declarations can be left implicit.

To be allowed to write a Fränkel operator, the types of the variables involved need to *widen* to a Mizar type that has the form `Element of` $A$. Only then can Mizar know that the defined set is really a set. Because else you could write proper classes like:

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

But in Mizar this expression is illegal because `set` doesn't widen to a type of the form `Element of`.

**Exercise 2.1.1** Write in Mizar syntax the statement that the set of prime numbers is infinite. Prove it.

## 2.2 Beyond first order: schemes

The proof of theorem `Th5` from `quantal1.miz` uses the `from` justification.

Mizar is based on first order predicate logic. However first order logic is slightly too weak to do ZF-style set theory. With first order logic ZF-style set theory needs infinitely many axioms. That's because ZF set theory contains an *axiom scheme*, the axiom of replacement.

Here is a table that shows how `scheme` and `from` is similar to `theorem` and `by`:

| *order* | *item* | *used* |
|---|---|---|
| first order | `theorem` | `by` |
| higher order | `scheme` | `from` |

Schemes are higher order but only in a very weak sense. It has been said that Mizar schemes are not second order but 1.001th order.

Let's study the first `from` that occurs in the proof. It justifies an equality:

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

The justification refers to the statement:

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

The scheme that is used here is:

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

The statement that this scheme proves has been underlined. It clearly is the same statement as the one that is being justified in the example. The general structure of a scheme definition is:

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

In the case of the `FRAENKEL:sch 5` scheme, the parameters are `B`, `F`, `G` and `P`. The first three are functions (those are written with round brackets), and the last is a predicate (written with square brackets).

To use a scheme you have to define *macros* with `deffunc` and `defpred`, that match the parameters of the scheme. In macro definitions the arguments to the macro are written as `$1`, `$2`, ... In this specific case the macros that are defined are:

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

(Apparently `B()` there doesn't need to be a macro. So Mizar can figure out by itself that in this specific example `B()` has to be instantiated with `carrier of Q`.)

For another example of a scheme let's look at the most commonly used scheme, which is `NAT_1:sch 1`. It allows one to do induction over the natural numbers. This scheme is:

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

So let's show how to use this scheme to prove some statement about the natural nummbers. For example consider the non-negativity of the natural numbers:

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

Clearly to prove this using the induction scheme, we need to define the following macro:

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

And then, the structure of the proof will be:

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

In the statement that is justified and the arguments to the scheme, the macros really have to be present (else Mizar won't know how to match things to the statements in the scheme), so you cannot write `0 >= 0` after `A1`, but everywhere else it doesn't matter whether you write `P[k]` or `k >= 0`, because almost immediately the macro `P[k]` will be expanded to its definition.

**Exercise 2.2.1** Write the natural deduction rules as Mizar schemes and prove them. For instance the rule of implication elimination ('modus ponens') becomes:

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

**Exercise 2.2.2** The axioms of the Mizar set theory are in the article called `TARSKI`. It's called that way because it contains the axioms of a theory called Tarski-Grothendieck set theory, which is ZF set theory with an axiom about the existence of big cardinal numbers. The `TARSKI` article contains one scheme:

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

Is it possible to prove this scheme using the Fränkel operator from the previous section (and without using other schemes from the MML)?

Conversely, is it possible to eliminate the use of the Fränkel operator from Mizar proofs by using the `TARSKI:sch 1` scheme?

## 2.3 Structures

The proof of theorem `Th5` from `quantal1.miz` uses the exciting type:

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

In this type `left-distributive`, `right-distributive`, `complete`, `Lattice-like` and `empty` are just attributes like before, but the mode `QuantaleStr` is something new.

The mode `QuantaleStr` is a *structure*. Structures are the records of Mizar. They contain *fields* that you can select using the construction:

`the` *field* `of` *structure*

An example is the expression:

```
the carrier of Q
```

In this the structure is `Q` and the field name is `carrier`. Many Mizar structures have the `carrier` field.

The definition of the `QuantaleStr` structure is:

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

Apparently it has four fields: `carrier`, `L_join`, `L_meet` and `mult`.

The modes `LattStr` and `HGrStr` are called the *ancestors* of the structure. They are structures that have a subset of the fields of the structure that is defined. Any term that has type `QuantaleStr` will automatically also get the types `LattStr` and `HGrStr`.

The definitions of the ancestors of `QuantaleStr` are:

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

and:

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

Structures are a powerful concept. They give the MML its abstract flavor.

**Exercise 2.3.1** Find the definition of the mode `Field` in `VECTSP_1`. What is the structure it is based on? Recursively trace all ancestors of that structure. Draw a graph of the widening relation between those ancestors.

**Exercise 2.3.2** The most basic structure in the MML is from `STRUCT_0`:

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

Is an object of type `1-sorted` uniquely determined by its `carrier`? Why or why not?

**Exercise 2.3.3** You can construct an object of type `1-sorted` as:

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

Use this notation to define functors `in` and `out` that map objects of type `set` to their natural counterparts in `1-sorted` and back. Prove in Mizar that `out` is a left inverse to `in`.

**Exercise 2.3.4** Are structures ZF sets or not? More specifically: if `Q` is a structure, can there be an `x` with `x in Q`?

If not, why not?

If so, can you prove the following?

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

If a structure has elements: is it uniquely determined by its elements?

**Exercise 2.3.5** Because of Gödel's theorems we know that Mizar's set theory is incomplete. That is, there has to be a Mizar formula $\phi$ without free variables, for which no Mizar proof exists of $\phi$, and also no Mizar proof exists of `not` $\phi$.

Can you think of a non-Gödelian $\phi$ that is provable nor disprovable in Mizar?