University of Alberta
Department of Computing Science
CMPUT 272

Introduction to Logic in Computing Science

H. James Hoover
Piotr Rudnicki

September 4, 2002

Course Overview

This course is based on the following philosophical principle:

All computing scientists must be able to employ rigorous reasoning to justify their claims about the behaviour of the programs they produce.

Thus the goal of this course is to introduce you to the logical and mathematical tools for reasoning about algorithms. Being able to effectively use these tools makes the difference between being a programming hack and a computing scientist. However, a course such as this can only examine the elementary material.

We will be covering in depth a small number of essential ideas. The course contains three main parts:

1. *Reasoning about programs.* Throughout the course we will be looking at programs as formal objects. Our goal will be to precisely specify the behaviour we wish our programs to have, and then to develop programs and prove that they have the desired behaviour.

2. *A formal system for predicate logic.* In this part we will introduce and use a formal system for predicate logic. Unlike previous versions of this course, and many courses on logic, we will not be using a system that exists only on paper. Instead we will be writing proofs that can be checked for correctness using the Mizar MSE system. In much the same way that you write a program, you will be entering your proofs into files and running them through Mizar. By the end of this part of the course you should be very comfortable with the idea of a proof, how to read one, and to some extent, how to write one.

3. *Basics: Sets, Relations, Functions, and Induction.* This could have been the first part of the course. We will introduce as required, various concepts such as sets, relations, functions, types, induction, and basic number theory.

The Format of These Notes

Previous editions of these notes were printed in a somewhat unorthodox way — they were single sided with the text on the left hand page. This was to encourage students to write their lecture notes on the right hand blank page beside the material we are discussing in the lecture. We even apologized to the left-handed students.

To save trees the notes are now printed double sided and three-hole punched. Now you can insert blank pages wherever you wish.

Many of the examples are preceded with a marginal note which gives the name of the file containing the actual text of the example. This saves you the trouble of entering in the example by hand in the event that you want to play with it.

Contents

0 Motivating Problems

Tweedledee (to Alice):
I know what you are thinking about, but it isn't so, nohow.
Tweedledum:
Contrariwise, if it was so, it might be;
if it were so, it would be;
but as it isn't, it ain't.
That's logic.

0.0 Why?

Many of our students complain that they see no reason for taking this course as part of the computing science curriculum. And despite our patient[1] explanations many students remain unconvinced[2]. So we would like to begin with some indirect motivation.

This section presents a number of puzzles, *all* of them relevant to a problem in computer science (but you may not see it at first glance). We hope that after taking this course you will be able to solve these problems to your satisfaction. The answers are not included, but you can always ask your instructor, or search the library. Each of these problems has been tackled more than once in the published literature.

The level of difficulty of the problems varies. Some are easy, almost everybody gets the solution in a couple of minutes. Some are difficult, and unless you know some specific techniques it is unlikely you will ever solve them. Some problems are actually open, people still argue what the solution should be. We have deliberately not marked the difficulty levels — it has been noted by many that knowing that a problem is difficult is a hindrance in solving it. Also, when someone gives you an easy puzzle, stresses the fact that the puzzle is really easy, yet you have no idea of how to solve it, this tends to make you nervous. We tried to avoid such situations.

0.1 Diamond in a box

0.1.0 3-box lottery

A genie (say you are dreaming) presents three boxes to you and states that one of them contains a diamond. The genie wants to keep the diamond, but says that you can have the diamond if you correctly guess which box contains it. You make a guess, but before you open the chosen box, the genie opens one of the remaining boxes that of course is empty. You are now allowed to change your original choice in order to maximize your chances to get the diamond. What would you do? Why?

[1]Well, sometimes impatient.

[2]To appreciate our perspective, ask yourself how you would convince a group of 6 year old kids that learning reading and arithmetic is good for them. This kind of explanation is usually not included in the grade 1 curriculum, although perhaps it should.

0.1.1 3-old boxes lottery

Another genie (say you are dreaming again) presents three boxes to you and tells you that one of them contains a diamond. You can have the diamond if you correctly guess which box contains it. You decide to have a closer look at the boxes before making a guess. After a closer examination of the boxes you discover that there is a statement on each of them printed in a small print as follows:

The diamond is not here.	The diamond is here.	The diamond is not here.

Which box do you choose?

0.1.2 2 boxes puzzle

This time the genie presents you with 2 boxes, one of which contains the diamond. Before letting you to make the choice, the genie explains that there are 2 packers who pack boxes: Jim and Peter. Whenever Jim packs a box he puts a true statement on the box, and whenever Peter packs a box he puts a false statement on the box. You do not know which box was packed by whom. Here are the boxes.

The diamond is not here.	Exactly 1 of the boxes was packed by Peter.

Where is the diamond?

0.1.3 Weird 2 boxes puzzle

The genie presents 2 boxes to you and tells that one of them contains a diamond.

The diamond is not here.	Exactly 1 statement on the boxes is false.

Now you reason this way. If the statement on the right box is true then the statement on the left box is false therefore the diamond is in the left box. If the statement on the right box is false then the statements on both boxes are false or both are true. They cannot be both true as the second is false. So both are false, then again the diamond must be in the left box. Therefore you choose the left box and open it and to your astonishment the box is empty. As you are convinced that your reasoning was flawless you feel cheated. But the genie claims he did not lie to you.

What is wrong?

Remark. The difference between this and the previous puzzle is quite subtle but also basic and essential for the rigorous treatment of logic.

0.2 Getting directions

0.2.0 Save your life

In yet another of your dreams you are in a prison on a remote planet. The puzzle-loving ruler of the planet puts you into a cell with 2 doors, one of which leads back to your spaceship “Enterprise”, the other one directly to a cage with a beast for which human meat is a much desired delicacy. There is a guard in front of each door. The guards are able to answer only “Yes” or “No” to any question you ask. Furthermore, one of them always tells the truth the other one always lies. Of course, you do not know who is who.

You are allowed to choose one of the guards and ask her exactly one question and then choose the door to meet your destiny.

What question do you ask?

0.2.1 Albanian food

One Sunday afternoon you develop a craving for Albanian food, and so head off along Whyte Avenue searching for such a restaurant. To your astonishment, you discover two of them, one across the street from the other — “The people’s choice”, and “The worker’s favourite”. You stop a biker standing outside the Princess Theatre and ask them which one is the better restaurant.

They reply that they do not know, but any of the trendy yuppies on the street does, since they are famous for their bizarre culinary tastes. But, the person tells you, the yuppies never tell the truth. Furthermore, do not expect more than a ‘Yes’ or ‘No’ answer from a yuppie.

You approach a Gucci-clad yuppie and ask them a question. From their answer, you know which restaurant is better. What question did you ask?

0.2.2 Albanian food II

After some time, the story repeats itself, except that you realize that the reputation of a restaurant is not fixed forever. You encounter a person who appears to have good taste. The problem is that you don’t know if they are a biker or a yuppie. Can you determine which restaurant is better?

0.2.3 Albanian food III

Suppose that the person you talked to originally was not a true biker, but in fact a yuppie pretending to be a biker. Can you determine which restaurant is better?

0.3 Troubles with existence

0.3.0 Unsocial uncle

An uncle of Peter chooses his friends by the following rule: he does not like people who like themselves and he likes people who do not like themselves.

Does he like himself?

Well, if he likes himself then he does not like himself. If he does not, then he likes himself.

So what should the uncle do?

0.3.1 God exists

(The argument of St. Anselm) God is a supreme being and no being is superior to God. The atheists, who deny that God exists must be able to conceive of something superior to God in order to know what they deny. But nothing superior to God can be conceived of. Therefore God exists.

(The argument of Descartes) By definition God is a being which has all properties. Hence, by definition, God must also have the property of existence. Therefore God exists.

If you are convinced by the above arguments then consider the following question: If the tail of a dog was called a leg, how many legs would a dog have? Abraham Lincoln answered this question: "Four; calling the tail a leg doesn't mean that it is one."

0.3.2 Irresistible and Immovable

By an irresistible cannonball we shall mean a cannonball which knocks over everything in its way. By an immovable post we shall mean a post which cannot be knocked over by anything.

What happens if an irresistible cannonball hits an immovable post?

0.3.3 Mother Eve

Theorem. There is a woman on Earth such that if she becomes sterile, the whole human race will die out because all women will become sterile.

Proof. Either all women will become sterile or not. If yes, then any woman satisfies the theorem. If no, then there is a woman that does not become sterile. She is then the one, such that if she becomes sterile (but she does not) then the human race will die out.□

Is the above convincing?

0.3.4 Santa Claus exists

You are presented the following piece of paper:

> IF THE STATEMENT ON THIS PIECE OF PAPER IS TRUE
> THEN SANTA CLAUS EXISTS.

Clearly, if the statement is true, then it follows that Santa Claus exists. Hence the statement is true and if the statement is true then Santa Claus exists.

Therefore Santa Claus exists.

Are you convinced? In any case read the following variation of the above argument:

Wife: Santa Claus exists, if I am not mistaken.

Husband: Well of course Santa Claus exists, *if you are not mistaken.*

Wife: Hence my statement is true.

Husband: Of course!

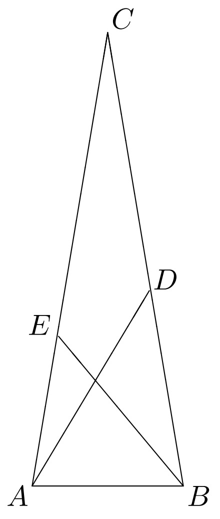
Wife: So I was not mistaken — and you admitted that if I am not mistaken, then Santa Claus exists. Therefore Santa Claus exists.

Such outlandish conversations are often heard among computing scientists.

0.4 Geometric brain teasers

0.4.1 An isosceles triangle

Consider the following triangle:



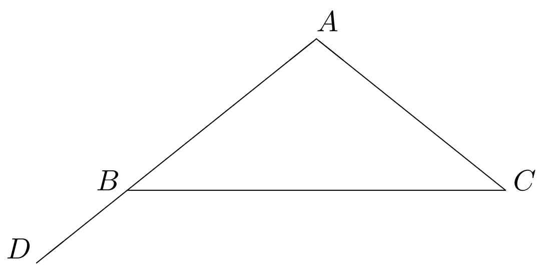
Angle measures:
$\angle ACB = 20°$
$\angle ABC = \angle BAC = 80°$
$\angle BAD = 60°$
$\angle ABE = 50°$

What is the measure of $\angle ADE$?

Remark. This problem is completely uninteresting if one employs trigonometry.

0.4.0 Another isosceles triangle

Consider the following figure:



Given:
$\angle ABC = \angle BCA = 40°$
$\overline{AD} = \overline{BC}$

What is the measure of $\angle ADC$?

0.5 What is the colour of your hat?

0.5.0 3 white and 2 black hats

Three men, name them A, B, and C entered a hat shop in which there were 3 white hats and 2 black hats. The men stood in a row such that C saw both A and B, B saw A but did not see C, and A saw neither B nor C. The shop assistant put some hats on their heads while they kept their eyes closed (we assume our men are honest) and the remaining hats were hidden behind the counter. After they open their eyes they tried to guess the colours of their hats. Here is what they said.

C: I do not know the colour of my hat. B: I also do not know the colour of my hat. A: If both of you do not know, then I know the colour of my hat.

What was the colour of A's hat?

0.5.1 At least one white hat

The same three gentlemen entered another hat shop stocked with a practically infinite number of white and black hats. They close their eyes and the shop assistant put some hats on their heads. They knew that at least one of the hats was white. This time they stood in such a way that everybody could see anybody else's hat, but not his own.

Now, they open their eyes and the shop assistant asks them the same question, one at a time in a round-robin fashion, "Do you know the colour of your hat?" keeping the answer secret. After obtaining the answers from all three of them, the shop assistant reveals that the answers were:

A: No. B: No. C: No.

Now the shop assistant starts the second round of asking the same question as in the first round. Again, answers are kept secret until all the answers have been collected and then the shop assistant announces that the answers were:

A: Yes. B: Yes. C: No.

What were the colours of their hats?

0.5.2 At least one white hat and no questions asked

The same three gentlemen enter yet another hat shop stocked with a practically infinite number of white and black hats. But this time they came on a practical purpose. They wanted a job and there was an opening for the shop assistant in the very shop. The shop owner decided to choose one of them for the job using the following procedure.

The lights were switched off and in complete darkness the owner put a hat on each of their heads at least one of which was white. The applicants stood in such a way that everybody could see anybody else's hat but not his own. The lights were turned on and the owner said that the first to guess the colour of his hat gets the job.

After a long period of silence B shouted "My hat is white!".

Did he get the job?

Remark. This puzzle is substantially different from the previous one. It touches on a quite interesting problem in (computer) communications.

0.6 1,000,000

Write a program that prints a statement in plain English about a natural number n that is true for, and only true for, all values of n less than one million.

0.7 Daemon in a pentagon

There is pentagon and at each vertex there is an integer number. The numbers can be negative but their sum is positive. A daemon living inside the pentagon manipulates the numbers as follows. If it spots a negative number at some vertex it performs the following atomic action: it adds the number to the numbers residing at neighbouring vertices and then negates the number at the original vertex.

Prove that no matter what numbers are given to the devil, after some time the devil will not be able to change any of the numbers.

Part I
Reasoning about programs

Promise: When you read these pages at this point of your education, you my find the material impractical, toyish, and useless. If this is your opinion, then we would like to ask you to read the stuff again just before you graduate. If your opinion does not change we will do something to refund your money.

1 (Small) Programming problems

This section presents a sequence of small programming problems. Most of them are completely solved. Some are not finished and are left as exercises.

The treatment of the examples is far from formal, however, it is rigorous and detailed. The formal treatment of the same material would require us to use mathematical machinery that would intimidate a beginner.

The programs are written in a notation that does not have an interpreter implemented on a computer; therefore you cannot communicate the programs to a machine and as a consequence you cannot run and test the programs. However, you can use the notation to write algorithms that you would like to communicate to your friend or just to make a record of for future use.

1.1 On the programming notation

We assume the reader is familiar with a programming language (preferably Pascal).

1.1.0 Data types

Boolean	$\{false, true\}$,
Int	integers,
Nat	natural numbers,
Rat	rational numbers.
array	the array can be of any type, indexed by any finite subrange of integers, see examples for syntax.

1.1.1 Expressions

Expressions are built of variables and constants with the help of common operators (such as $+$ or $\cdot$), and are therefore not described here. The syntax of expressions, and the standard functions used are basically the same as in the programming language Pascal. The notable difference is that instead of **mod** we use **rem**, the difference is discussed in section 1.12.

Evaluating an expression must not have any side effects.

All variables used in expressions must be previously declared.

1.1.2 Formulas

The preconditions, invariants, and postconditions are written as logical formulae using various species from the zoo of mathematical symbols.

1.1.3 Commands

Semicolons separate commands, however a semicolon may be skipped after a keyword.

Do nothing command

The command **skip** does not affect a computation in any way, it is sometimes needed for syntactic reasons. An empty command could have been used but **skip** stresses that there is nothing to do in a particular place of a program.

Assignment

We use := as the assignment operator. The assignment operator is multiple in the following sense: the assignment

$$x, y := x + y, x * y$$

is executed as follows:

(a) $x + y$ and $x * y$ are evaluated, let their value be *t1* and *t2*, respectively,

(b) x and y are assigned the values $t1$ and $t2$, respectively.

This ensures that all calculations on the right hand side of := are finished before the values of the variables on the left hand side are altered.

Alternative command

```
if
  ▯ $guard_1 \longrightarrow command_1$
  ▯ $guard_2 \longrightarrow command_2$
              . . .
  ▯ $guard_n \longrightarrow command_n$
fi
```

where each *guard$_i$* must be a boolean expression, is executed as follows:

(a) All guard expressions are evaluated. If

$$\textbf{not}\ (guard_1\ \textbf{or}\ guard_2\ \textbf{or} \cdots \textbf{or}\ guard_n)$$

then the result of the alternative command is undefined and the execution of a program containing such a command would be aborted (but we do not run our programs). When writing a program you have to prove that this does not happen. Note that this includes also proving that the values of all guards can be successfully computed (that is without runtime errors like dividing by *0* or referring to a non-existing element in an array).

(b) Exactly one command is chosen from among those whose guard expression evaluated to true, and executed. This completes the execution of the alternative command. Note: If more than one guard evaluates to true, then an arbitrary choice of which of the corresponding commands to execute is made. We will discuss the use of overlapping guards in section 1.3.

Iterative command

```
do
    Variant: The variant expression
    Invariant: The invariant formula
  ▯ guard_1 ( $\xrightarrow{\textbf{exit}}$ | $\longrightarrow$ command_1 )
  ▯ guard_2 ( $\xrightarrow{\textbf{exit}}$ | $\longrightarrow$ command_2 )
        . . .
  ▯ guard_n ( $\xrightarrow{\textbf{exit}}$ | $\longrightarrow$ command_n )
od
```

It is required that at least one of the guards is followed by $\xrightarrow{\textbf{exit}}$. The *variant expression* must be an expression of type *Nat*.

The iterative command is executed as follows: First, the *variant expression* is evaluated and the value is put into variable *ve_value* which is inaccessible to the programmer. We must prove that this value is non-negative. Then

(a) If the *invariant formula* is false the hypothetical execution of the program containing the iterative construct would be aborted. When writing a program you have to prove that this does not happen. Otherwise,

(b) All guard expressions are evaluated. If

$$\textbf{not}\ (\textit{guard}_1\ \textbf{ or }\ \textit{guard}_2\ \textbf{ or} \cdots \textbf{or}\ \ \textit{guard}_n)$$

then the result of the iterative command is undefined and the execution of a program containing such a command would be aborted. When writing a program you have to prove that this does not happen. As in the case of the alternative command, one has to prove also that the evaluation of all guards terminates without runtime errors.

(c) Of all the guards that evaluated to true, one is chosen non-deterministically (by flipping a coin, or rolling a dice, or always picking the first, or some other unknown mechanism) and then

  (c1) If the chosen guard is followed by $\xrightarrow{\textbf{exit}}$, then the execution of the iterative command is terminated, otherwise

  (c2) the corresponding command is executed and then,

  (c3) the *variant expression* is evaluated. If the obtained result is non-negative but smaller than the current value of *ve_value*, it becomes the new value of the *ve_value* variable and the iterative command continues its execution at (a) by examining the invariant, otherwise

  (c4) the result of the iterative command is undefined and the execution of a program containing such a command would be aborted. When writing a program you have to prove that this does not happen.

1.1.4 Specification

Syntactically, a specification is built as follows:

> *Declarations of variables*
>
> **Preconditions**: The *preconditions formula.*
>
> $\boxed{?}$
>
> **Postconditions**: The *postconditions formula.*

The syntax of formulae will be given later, but we will use formulae long before all the details are explained. When writing formulae we make use of common mathematical notation, which is not explained in these notes.

1.1.5 Program

Syntactically, a program is built as follows:

> *Declarations of global variables*
>
> **Preconditions**: The *preconditions formula.*
>
> *Declarations of local variables*
>
> *Code: a sequence of commands*
>
> **Postconditions**: The *postconditions formula.*

1.2 swap

Point: Straight line programs are easy to understand.

The following specification calls for a program that swaps the values of integer variables a and b.

> *Int* a, b;
> **Preconditions**: $A = a \;\&\; B = b$
> $\boxed{?}$
> **Postconditions**: $a = B \;\&\; b = A$

Note how we used A and B to name the initial values of the variables a and b. The required program is simple to find:

> *Int* *temp*;
> $temp := a$;
> $a := b$;
> $b := temp$

How do we argue that the code satisfies the stated specification? We have to know how to reason about the simple command of assignment and then about a sequence of assignments. We will do it in a backward fashion. First, we ask the question: under what conditions does the last assignment establish the desired postconditions?

The assignment $b := temp$ may change the value of b. The final postconditions state the requirements for the final value of b. So, in order to know something about the value of b after the assignment, we better know the same thing about *temp* before the assignment was made. We express this by the following correct program (declarations of variables are omitted):

> **Preconditions-3**: $a = B \;\&\; temp = A$
> $b := temp$
> **Postconditions**: $a = B \;\&\; b = A$

The $b := temp$ command establishes the original postconditions if it is executed in a state that satisfies the formula labelled **Preconditions-3**. The question is now, how to establish these conditions? Answer: these conditions must be established by the previous instruction! Which preconditions must be met for $a := b$ to establish $a = B \;\&\; temp = A$? Since we have to know something about a after the assignment we better know the same about b before the assignment is done. We express this by the following correct program:

> **Preconditions-2**: $b = B \;\&\; temp = A$
> $a := b$
> **Postconditions-2**: $a = B \;\&\; temp = A$

Now, under what conditions does the assignment $temp := a$ establish $b = B \;\&\; temp = A$? For reasons similar to those discussed above we see that the following program is correct:

```
    **Preconditions-1**: $b = B \;\&\; a = A$
    $temp := a$
    **Postconditions-1**: $b = B \;\&\; temp = A$
```

But our original **Preconditions** mean the same as the formula labelled **Preconditions-1**. Note that:

$$a = A \;\&\; b = B \textbf{ iff } b = B \;\&\; a = A$$

What does it mean altogether?

- The original **Preconditions** guarantee **Preconditions-1**.
- The first assignment establishes **Postconditions-1**.
- As **Postconditions-1** mean the same as **Preconditions-2**, the second assignment establishes **Postconditions-2**.
- As **Postconditions-2** are identical to **Preconditions-3**, the third assignment establishes the original **Postconditions**.

There is a silly-looking but important question: how would you like to have your program presented?

Is something that hides all the details of our reasoning satisfactory? Look at this:

```
$Int\ \ a, b$;
    **Preconditions**: $A = a \;\&\; B = b$
    $Int\ \ temp$;
    $temp := a$;
    $a := b$;
    $b := temp$
    **Postconditions**: $a = B \;\&\; b = A$
```

Or, should we display our reasoning by talking about a single command at a time, like we did earlier, displaying all the numbered pre- and post-conditions?

Or, would you prefer that some of our reasoning is shown by stating explicitly the intermediate conditions during the computation, e. g. like this (read it forward):

```
$Int\ \ a, b$;
    **Preconditions**: $A = a \;\&\; B = b$
    $Int\ \ temp$;
                **Conditions** : $b = B \;\&\; a = A$
    $temp := a$;
                **Conditions** : $b = B \;\&\; temp = A$
    $a := b$;
                **Conditions** : $a = B \;\&\; temp = A$
    $b := temp$
    **Postconditions**: $a = B \;\&\; b = A$
```

1.3 min and max

Point: How to reason about an alternative command?

The following specification calls for a program that assigns the maximum of integers a and b to r:

```
Int  a, b;
       Preconditions:
Int  r;
       $\boxed{?}$
       Postconditions: $((r = a$ or $r = b)$ & $r \geq a$ & $r \geq b)$
```

The specification strongly suggests that an **if** instruction be used in order to establish the postcondition by first deciding which is larger the value of a or the value of b, and then assigning the larger value to r.

```
if
  ▯ $a \geq b \longrightarrow r := a$
  ▯ $b \geq a \longrightarrow r := b$
fi
```

When the guard is true, then the chosen command establishes the postcondition as can be seen from the discussion below. The following two pieces of code are correct by the definition of the := operator:

```
Piece1:
       Preconditions:
       $r := a$
       Postconditions: $r = a$
Piece2:
       Preconditions:
       $r := b$
       Postconditions: $r = b$
```

Let us note that the $r := a$ command with the precondition $a \geq b$ establishes the following postconditions:

```
Preconditions: $a \geq b$
$r := a$
Postconditions: $r = a$ & $r \geq b$
```

In propositional logic with equality we can prove that

$$r = a \,\&\, r \geq b \textbf{ implies } ((r = a \textbf{ or } r = b) \,\&\, r \geq a \,\&\, r \geq b)$$

Combining the last two observations we get:

Preconditions: $a \geq b$
$r := a$
Postconditions: $((r = a$ **or** $r = b)$ & $r \geq a$ & $r \geq b)$

Similar reasoning applies to the *Piece2* of code (the reader is encouraged to reconstruct the reasoning as an exercise) yielding:

Preconditions: $b \geq a$
$r := b$
Postconditions: $((r = a$ **or** $r = b)$ & $r \geq a$ & $r \geq b)$

We will now argue that the alternative command establishes the desired postconditions of the initial specification. Note what follows:

- At least one of the guards is always true, $a \geq b$ **or** $b \geq a$, and exactly one of the guarded commands is executed.
- If a guard, $a \geq b$ or $b \geq a$, is true, then the corresponding guarded command—$r := a$ or $r := b$, respectively— establishes the postconditions as shown by the above analysis.

Note that when $a = b$, then the alternative command may behave non-deterministically, that is, we do not know which of the guarded commands is selected for execution. This does not matter for correctness: the desired postconditions will be established whichever is selected.

Problem: Specify, find the program, and argue for the correctness of an analogous code that finds the minimum of two integers.

1.4 1-bit adder

Points: Logic relates to hardware.
Do not write alternative commands if you do not have to.

Proficiency in manipulating propositional formulae is important for computation design. Consider the following specification:

```
Nat  a, b;
     Preconditions: $a, b \in \{0, 1\}$;
Nat  c, s;
     $\boxed{?}$
     Postconditions:
          ($a = 0$ & $b = 0$ implies $c = 0$ & $s = 0$)  &
          ($a = 0$ & $b = 1$ implies $c = 0$ & $s = 1$)  &
          ($a = 1$ & $b = 0$ implies $c = 0$ & $s = 1$)  &
          ($a = 1$ & $b = 1$ implies $c = 1$ & $s = 0$)
```

The specification is complete as it specifies the output values for each possible initial setting of the input parameters. How do we develop the needed program? One may attempt to write the program by directly interpreting the postconditions. This would replace the $\boxed{?}$ with the following **if** statement:

```
if
  [] $a = 0$ & $b = 0$ $\longrightarrow$ $c, s := 0, 0$
  [] $a = 0$ & $b = 1$ $\longrightarrow$ $c, s := 0, 1$
  [] $a = 1$ & $b = 0$ $\longrightarrow$ $c, s := 0, 1$
  [] $a = 1$ & $b = 1$ $\longrightarrow$ $c, s := 1, 0$
fi
```

Such a solution is bad as we test the same thing more than once, which makes the program long and ugly. We may try to change the program but first let us have a closer look at the specification. From our postconditions we can infer (among other useless facts) the following:

not $c = 0$ **iff** $c = 1$,
not $s = 0$ **iff** $s = 1$,
$a = b$ **iff** $s = 0$,
$a = 1$ & $b = 1$ **iff** $c = 1$.

This observation suggests setting c and s separately:

```
**if**
  ▯ $a = b \longrightarrow s := 0$
  ▯ $a \neq b \longrightarrow s = 1$
**fi**;
**if**
  ▯ $a = 1 \; \& \; b = 1 \longrightarrow c := 1$
  ▯ $a \neq 1$ **or** $b \neq 1 \longrightarrow c := 0$
**fi**
```

The above program can be transformed to save on writing and execution time:

```
**if**
  ▯ $a = b \longrightarrow c, s := a, 0$
  ▯ $a \neq b \longrightarrow c, s := 0, 1$
**fi**
```

Our original problem can be transformed into a more interesting one. All the values that are used in our problem are *0* or *1* as we are trying to build a 1-bit adder. Consider four *Boolean* variables A, B, C, and S which are defined in terms of our integer variables as follows:

A **iff** $a = 1$,
B **iff** $b = 1$,
C **iff** $c = 1$,
S **iff** $s = 1$.

With this translation we can now rewrite the postconditions as follows:

Postconditions:

$$\begin{array}{llclclclcl}
(& \textbf{not}\ A & \& & \textbf{not}\ B & \textbf{implies} & \textbf{not}\ C & \& & \textbf{not}\ S &) & \& \\
(& \textbf{not}\ A & \& & B & \textbf{implies} & \textbf{not}\ C & \& & S &) & \& \\
(& A & \& & \textbf{not}\ B & \textbf{implies} & \textbf{not}\ C & \& & S &) & \& \\
(& A & \& & B & \textbf{implies} & C & \& & \textbf{not}\ S &) &
\end{array}$$

If we now compare the above with the definitions of the logical connectives conjunction and equivalence we can observe that:

C **iff** $A \; \& \; B$,
S **iff not** $(A$ **iff** $B)$.

But this effect can be achieved in Pascal by single assignments:

```
$C$ := $A$ **and** $B$;
$S$ := **not** $(A = B)$
```

Note how this solution is short and elegant!

How can it help us in solving the original problem? If we are in a Pascal-like environment, we will have to do some pre- and post-processing in order to translate first from integers to Booleans and then back. This backward translation is achieved by employing the standard Pascal function *ord*. (Remember that the type *Boolean* is predefined in Pascal as (*false*, *true*).) The initial $\boxed{\textbf{?}}$ is replaced by:

```
Boolean  A, B, C, S;
                                        {Translate from integers to Booleans.}
    A := a = 1; B := b = 1;
                                               {Do the work on Booleans.}
    C := A and B;
    S := not (A = B);
                                        {Translate from Booleans to integers.}
    c := ord(C); s := ord(S)
```

Reminder for the C language speakers: in the programming language C, no pre- or post-processing is required.

Question: If A **iff** B is expressible in Pascal simply as $A = B$, how would you express A **implies** B?

1.5 The Coffee Can Problem

Points: **Variants** are used to demonstrate the termination of iterative processes.
Invariants are used for proving what iterative processes do.

Problem: A coffee can contains some black beans and white beans. The following process is repeated as long as possible:

> Randomly select two beans from the can. If they have the same colour, throw them out, but put another black bean in. (Enough extra black beans are available to do this.) If they are different colours, place the white one back into the can and throw the black one away.

Prove that the process stops with exactly one bean in the can. Specify the colour of the final bean in terms of the number of white beans and the number of black beans initially in the can.

Exercise: The above is an informal specification of a process. Are you sure how the process is run?

Let us write a program that describes the behaviour of a robot that manipulates coffee beans.

The coffee can

```
Nat  $W, B, w, b$;
     Preconditions: $W > 0$ & $B > 0$;
Bean  $bn1, bn2$;
     $w := W; b := B$;
     do
          Variant: $w + b$.
          Invariant: $w =_{\text{mod } 2} W$.
       ▯ $w + b = 1 \xrightarrow{\text{exit}}$
       ▯ $w + b > 1 \longrightarrow bn1 := PickaBean;\ bn2 := PickaBean;$
               if
                 ▯ $bn1.col = white$ & $bn2.col = white \longrightarrow$
                                 $w := w - 2;\ PlaceBlackBean;\ b := b + 1$
                 ▯ $bn1.col = black$ & $bn2.col = black \longrightarrow$
                                 $PlaceBlackBean;\ b := b - 1$
                 ▯ $bn1.col \neq bn2.col \longrightarrow b := b - 1;\ PlaceWhiteBean$
               fi
     od

     Postconditions:
          R0: $w + b = 1$,
          R1: $(W =_{\text{mod } 2} 0$ implies $b = 1)$ & $(W =_{\text{mod } 2} 1$ implies $w = 1)$.
```

End of the coffee can

The robot can perform the following operations:

PickaBean, PlaceBlackBean, PlaceWhiteBean

with obvious meaning. Note that *PickaBean* is non-deterministic: when executed twice with the same can contents it may give different results.

The preconditions say that at the start we have at least one white bean and at least one black bean.

Definition. Let a and b be natural numbers. $a =_{\text{mod } 2} b$ means

$$\exists\, z \in \mathit{Int} :: a - b = 2 \cdot z.$$

We will now argue that the program is correct, that is, when the program is run in a state satisfying the preconditions, it will terminate in a state satisfying the postconditions.

We have to prove a number of properties of our program.

The if command is correct, i.e. it never aborts. We have to prove that all guard expressions are correct boolean expressions and at least one of them always evaluates to *true*. We assume that inspecting the colour of a bean is always successful. Each of the picked up beans *bn1* and *bn2* must be either white or black and therefore the following sentence is always true:

$$\begin{array}{ll} & bn1.col = white \;\&\; bn2.col = white \\ \textbf{or} & \\ & bn1.col = black \;\&\; bn2.col = black \\ \textbf{or} & \\ & bn1.col \neq bn2.col \end{array}$$

The do command is correct. In order to prove this we have to demonstrate that:

- One of the guards of the iterative command is always true.
- The variant expression is always non-negative and decreases with every iteration.
- The invariant is always true.

Each of these three properties requires an *inductive* reasoning. In order to show inductively that some statement is always true about the iterative command we will prove:

1. *First time:* The statement is true when the loop is executed for the first time.
2. *Once through the loop after an arbitrary number of loop executions:* After assuming that the statement is true and the loop does not exit, we demonstrate that the statement is true when the loop is about to be executed the next time.

We will discuss other details of inductive reasoning later in the course.

The loop never aborts. We have to prove that

$$w + b = 1 \textbf{ or } w + b > 1$$

at the moment when the loop guards are evaluated. The above formula can be written as $w + b \geq 1$.

First time. From the preconditions we know that $W > 0$ and $B > 0$ and because they are natural numbers $W \geq 1$ and $B \geq 1$ which yields $W + B > 1$. When the loop is executed for the first time we have $w + b = W + B$, and thus $w + b \geq 1$.

Once through the loop. Let w_c and b_c be the *current* values of w and b. Assume the loop does not exit, i.e.

$$A\colon w_c + b_c > 1.$$

After the loop executes its body once, let the *next* values of w and b be w_n and b_n. Our goal is to prove that $w_n + b_n \geq 1$. How do the next values relate to w_c and b_c? This can be answered by considering three cases of the **if** command.

- Case $bn1.col = white$ & $bn2.col = white$.

  $w_n = w_c - 2$ and $b_n = b_c + 1$. Then $w_n + b_n = w_c + b_c - 1$. Because of A we have $w_n + b_n \geq 1$.

- Case $bn1.col = black$ & $bn2.col = black$.

  $w_n = w_c$ and $b_n = b_c - 1$. Then $w_n + b_n = w_c + b_c - 1$. Because of A we have $w_n + b_n \geq 1$.

- Case $bn1.col \neq bn2.col$.

  $w_n = w_c$ and $b_n = b_c - 1$. Then $w_n + b_n = w_c + b_c - 1$. Because of A we have $w_n + b_n \geq 1$.

We have proven that no matter which action of the iterative command is executed we have $w_n + b_n \geq 1$.

The two steps named *First time* and *Once through the loop* demonstrate that $w + b \geq 1$, which proves that the loop will never abort due to a failure in evaluating guards.

The loop terminates. This claim is proven with the help of the *variant* expression. Recall (page 11) that the variant expression must be a natural (integer, non-negative) expression which decreases every time the loop is executes. If so, the loop must terminate, as no natural expression can decrease forever. This is another inductive reasoning.

In our case the variant expression is claimed to be $w + b$. Without a proof that the expression is a variant of the loop such a claim is nothing more than a guess. Note also the a loop may have more than one variant expression (make sure that you understand this).

First time. Here we have to show that $w + b$ is a natural expression. It is easy to see that this is the case.

Once through the loop. Look to the previous *Once through the loop* for notation and assumptions.

We have to prove that $0 \leq w_n + b_n < w_c + b_c$. From the reasoning above we know that $w_n + b_n = w_c + b_c - 1$ and that $1 \leq w_n + b_n$. Clearly $w_c + b_c - 1 < w_c + b_c$.

Therefore we have proved that $w+b$ is the sought variant expression and we can conclude that the loop terminates. (Note: variant expression for a terminating loop is not unique, in the above case take for example $w + b + 31$. It is a variant expression as well.)

The invariant. We now turn to proving that the claimed invariant holds. We will use the same technique and notation as above.

First time. Because $w = W$, then $w =_{\mathrm{mod}\ 2} W$.

Once through the loop. (Look above for notation.) Assume the invariant holds, i.e.

$$I\colon w_c =_{\mathrm{mod}\ 2} W.$$

and the loop does not exit.

Our goal is to prove that $w_n =_{\mathrm{mod}\ 2} W$. We consider three cases of the **if** command.

- Case $bn1.col = white$ & $bn2.col = white$.

  $w_n = w_c - 2$ and therefore $w_n =_{\mathrm{mod}\ 2} W$ by I and the definition of $=_{\mathrm{mod}\ 2}$.

- Case $bn1.col = black$ & $bn2.col = black$.

  $w_n = w_c$ and therefore $w_n =_{\mathrm{mod}\ 2} W$ by I.

- Case $bn1.col \neq bn2.col$.

  $w_n = w_c$ and therefore $w_n =_{\mathrm{mod}\ 2} W$ by I.

Therefore the stated invariant is now proven. The invariant says that the parity of white beans in the can never changes.

One may ask why we picked this invariant and not another one. The answer is simple: this invariant helps us in proving the expected postconditions. But indeed, the loop has many invariants, some of them completely useless as for example the following property which is always true: $1 = 1$. Note that when proving that the loop never aborts, we have shown that $w + b \geq 1$ is an invariant of the loop.

Proving the postconditions. We know that the loop terminates. The question is now: do the postconditions hold when it does? The answer is *yes* because of the following reasoning.

When the loop exits (it must as it terminates) we have

$$R0\colon w + b = 1$$

as there is only one exit guard which says exactly this. We also know that the invariant holds as it held just before the loop exited.

$$Inv\colon w =_{\mathrm{mod}\ 2} W.$$

How can we prove the postcondition labelled *R1*? We have to prove two statements each being an implication. We will closely follow a formalism that we will discuss in detail later.

```
*R1a:* $W =_{\mathrm{mod}\ 2} 0$ **implies** $b = 1$
**proof**
    **assume** $W =_{\mathrm{mod}\ 2} 0$;
        $w \leq 1$ by *R0*, i.e. $w = 0$ or $w = 1$. We know from assumption that $W$ is
        even, then $w$ must also be even by *Inv.* Hence $w = 1$ is impossible, therefore
        $w = 0$, and because of *R0* we conclude
    **thus** $b = 1$

*R1b:* $W =_{\mathrm{mod}\ 2} 1$ **implies** $w = 1$ **proof**
    **assume** $W =_{\mathrm{mod}\ 2} 1$;
        $w \leq 1$ by *R0*, i.e. $w = 0$ or $w = 1$. We know from assumption that $W$ is
        odd, then $w$ must also be odd by *Inv.* Therefore $w = 0$ is impossible. And
        we conclude
    **thus** $w = 1$
```

The postcondition *R1* follows from *R1a* and *R1b.*

In words we can say: the process of removing beans from the can terminates with one bean in the can; the remaining bean is white exactly when the initial number of white beans was odd.

Consider the following variations of the coffee can problem.

Variation 1 : Change the **if** statement to the following:

```
if
  ▯ $bn1.col = white \ \& \ bn2.col = white \longrightarrow$
                          $w := w - 2;\ PlaceBlackBean;\ b := b + 1$
  ▯ $bn1.col = black \ \& \ bn2.col = black \longrightarrow PlaceBlackBean;\ b := b - 1$
  ▯ $bn1.col \neq bn2.col \longrightarrow w := w - 1;\ PlaceBlackBean$
fi
```

State a variant and an invariant of the changed loop. Find out the colour of the remaining bean by formulating the postcondition. Note: the preconditions have not changed.

Variation 2 : Change the **if** statement to the following:

```
if
  ▯ $bn1.col = white \ \& \ bn2.col = white \longrightarrow PlaceWhiteBean;\ w := w - 1$
  ▯ $bn1.col = black \ \& \ bn2.col = black \longrightarrow PlaceBlackBean;\ b := b - 1$
  ▯ $bn1.col \neq bn2.col \longrightarrow w := w - 1;\ PlaceBlackBean$
fi
```

State a variant and an invariant of the changed loop. Find out the colour of the remaining bean by formulating the postcondition. Note: the preconditions have not changed.

Variation 3 : (An ill defined process.)
Change the **if** statement to the following:

```
if
  ▯ $bn1.col = white \ \& \ bn2.col = white \longrightarrow PlaceWhiteBean;\ w := w - 1$
  ▯ $bn1.col = black \ \& \ bn2.col = black \longrightarrow PlaceBlackBean;\ b := b - 1$
  ▯ $bn1.col = white \ \& \ bn2.col = black \longrightarrow PlaceBlackBean;\ w := w - 1$
  ▯ $bn1.col = black \ \& \ bn2.col = white \longrightarrow PlaceWhiteBean;\ b := b - 1$
fi
```

State a variant of the changed loop.

In the previous cases it is possible to state the colour of the remaining bean knowing only the initial values of W and B. This is impossible now. Why? Give the shortest explanation you can.

1.6 Multiplication: slow and fast

Points: How to accumulate knowledge?
How to make your programs run faster?

The following program multiplies two integers using only addition. This is the slow multiplication.

The slow multiplication

```
Rat  $X, res$;
Nat  $Y$;
     Preconditions:
Nat  $y$;
     $y, res := 0, 0$;
     do    Variant: $Y - y$.
           Invariant: $X \cdot y = res$.
       ▯ $y = Y \xrightarrow{\textbf{exit}}$
       ▯ $y < Y \longrightarrow y, res := y + 1, res + X$
     od
     Postconditions: $X \cdot Y = res$.
```

End of the slow multiplication

We will now demonstrate that the claimed invariant of the loop is indeed always true. We will follow the form of inductive reasoning introduced in the coffee can example.

First time. When the loop body is to be executed for the first time we have: $y = 0$ and $res = 0$ which makes $X \cdot y = res$ true as $X \cdot 0 = 0$.

Once through the loop. Assume that after a number of loop executions the *current* values of y and res variables are y_c and res_c, respectively. Assume that the invariant is true, that is:

$$A: \quad X \cdot y_c = res_c.$$

We prove that if the loop body is executed once more without exiting the loop then the invariant will be true again. If the loop does not exit, the assignment $y, res := y+1, res+X$ is executed and the *next* values of y and res are $y_n = y_c+1$ and $res_n = res_c+X$, respectively. We have:

$$\begin{aligned} X \cdot y_n &= X \cdot (y_c + 1) & \\ &= X \cdot y_c + X & \cdot \text{ distributes over } + \\ &= res_c + X & \text{by } A \\ &= res_n & \end{aligned}$$

which proves that the claimed invariant is true again.

Using a similar format of reasoning prove that the loop never fails and that it terminates. Finally, prove the postconditions. Note how the invariant combined with the exit guard implies the postconditions.

It is instructive to look at a version of the multiplication in which y decreases instead of increasing.

The slow multiplication going down

```
Rat  $X, res$;
Nat  $Y$;
     Preconditions:
Nat  $y$;
     $y, res := Y, 0$;
     do     Variant: $y$.
            Invariant: $X \cdot y + res = X \cdot Y$.
       ▯ $y = 0 \xrightarrow{\textbf{exit}}$
       ▯ $y > 0 \longrightarrow y, res := y - 1, res + X$
     od
     Postconditions: $res = X \cdot Y$.
```

End of the slow multiplication going down

The multiplication problem admits a much faster solution if we can use some other operations besides addition. The problem would entirely disappear if we could use the multiplication operator that can take an arbitrary natural number as its second argument. It would be too easy. Assume that besides addition and subtraction, we can use also the operations of multiplying by 2 and integer division by 2. These operations are considered quite primitive as they correspond to shifting bits in a memory unit, left and right, respectively. Employing these two operations leads to the so called fast multiplication.

The fast multiplication

```
Rat  $X, res$;
Nat  $Y$;
     Preconditions:
Rat  $x$;
Nat  $y$;
     $x, y, res := X, Y, 0$;
     do     Variant: $y$.
            Invariant: $x \cdot y + res = X \cdot Y$.
       ▯ $y = 0 \xrightarrow{\textbf{exit}}$
       ▯ $odd(y) \longrightarrow y, res := y - 1, res + x$
       ▯ $even(y)$ & $y > 0 \longrightarrow y, x := y$ div $2, 2 \cdot x$
     od
     Postconditions: $res = X \cdot Y$.
```

End of the fast multiplication

Problem: Try to rewrite the fast multiplication with y going up instead of going down. Is the obtained solution any better than the program above?

1.7 Exponentiation: slow and fast

Points: How to accumulate knowledge?
How to make your programs run faster?

The following program may prove useful for you one day as many languages do not provide a built-in exponentiation operator. On the other hand, most program libraries will have an exponentiation routine.

It would be instructive for you to compare the multiplication programs from the previous section with the programs below.

The slow exponentiation

```
Rat X, res;
Nat Y;
        Preconditions:
Nat y;
        $y, res := 0, 1$;
        do      Variant: $Y - y$.
                Invariant: $X^y = res$.
          ▯ $y = Y \xrightarrow{\textbf{exit}}$
          ▯ $y < Y \longrightarrow y, res := y + 1, res \cdot X$
        od
        Postconditions: $X^Y = res$.
```

End of the slow exponentiation

It is instructive to look at a version of the exponentiation in which y decreases instead of increasing.

The slow exponentiation going down

```
Rat X, res;
Nat Y;
        Preconditions:
Nat y;
        $y, res := Y, 1$;
        do      Variant: $y$.
                Invariant: $X^y \cdot res = X^Y$.
          ▯ $y = 0 \xrightarrow{\textbf{exit}}$
          ▯ $y > 0 \longrightarrow y, res := y - 1, res \cdot X$
        od
        Postconditions: $res = X^Y$.
```

End of the slow exponentiation going down

The above programs are slow. In both cases we perform $Y - 1$ multiplications by X. We can do much better than that if we employ the operation of integer division by 2. Any binary computer can perform this operation really fast.

The fast exponentiation

```
Rat  $X, res$;
Nat  $Y$;
     **Preconditions**:
Rat  $x$;
Nat  $y$;
     $x, y, res := X, Y, 1$;
     **do**    **Variant**: $y$.
           **Invariant**: $x^y \cdot res = X^Y$.
       ▯ $y = 0 \xrightarrow{\textbf{exit}}$
       ▯ $odd(y) \longrightarrow y, res := y - 1, res \cdot x$
       ▯ $even(y)\ \&\ y > 0 \longrightarrow y, x := y\ \textbf{div}\ 2, x \cdot x$
     **od**
     **Postconditions**: $res = X^Y$.
```

End of the fast exponentiation

Problem: Try to rewrite the fast exponentiation with y going up instead of going down. Is the obtained solution any better than the program above?

1.8 Array summation

Point: Start with knowing nothing and accumulate your knowledge by doing something.

Given an array $A[1..n]$ of rational numbers. Assign a rational variable *sum* to the sum of the array elements.

```
___________________________________________________________Array summation
    Nat  $n$;
    Rat  $A[1..n]$;
    Rat  $sum$;
         Preconditions:
    Nat  $k$;
         $k, sum := 0, 0$;
         do
              Variant: $n - k$.
              Invariant: $(\sum i : 1 \le i \le k : A[i]) = sum$.
           ▯ $k = n \xrightarrow{\textbf{exit}}$
           ▯ $k < n \longrightarrow k := k + 1; sum := sum + A[k]$
         od
         Postconditions: $(\sum i : 1 \le i \le n : A[i]) = sum$.
__________________________________________________End of the array summation
```

Note: the loop invariant together with the exit guard imply the postconditions.

Exercise: Why cannot the assignment commands $k := k + 1; sum := sum + A[k]$ be written as a multiple assignment $k, sum := k + 1, sum + A[k]$?

1.9 Array minimum

Point: Start with knowing nothing and accumulate your knowledge by doing something.

Given an array $A[1..n]$ of rational numbers. Assign a rational variable *min* to the value of the minimal element of the array. There is no operation available for finding minimum of two or more numbers.

Array minimum

```
Nat  $n$;
Rat  $A[1..n]$;
Rat  $min$;

     Preconditions:
Nat  $k$;
     $k, min := 0, maxRat$;
     do
          Variant: $n - k$.
          Invariant: $(\mathbf{MIN}\ i : 1 \le i \le k : A[i]) = min$.

       [] $k = n \xrightarrow{\text{exit}}$
       [] $k < n \longrightarrow k := k + 1$;
                        if
                          [] $A[k] < min \longrightarrow min := A[k]$
                          [] $A[k] \ge min \longrightarrow$ skip
                        fi
     od

     Postconditions: $(\mathbf{MIN}\ i : 1 \le i \le n : A[i]) = min$.
```

End of the array minimum

Note: the loop invariant together with the exit guard imply the postconditions.

1.10 Linear search and binary search

Point: Accumulate your knowledge but do not compute things that you can infer.

This problem concerns a one dimensional array of integers $A[1..n]$. Consider the following program that searches the array A for the value of the integer variable x and finds the smallest k such that $A[k] = x$.

Linear search

```
Nat n;
Int A[1..n];
Int x;
    Preconditions: $\exists\, i : 1 \le i \le n : A[i] = x$.
Nat k;
    $k := 1$;
    do
        Variant: $n - k$.
        Invariant: $(\exists\, i : k \le i \le n : A[i] = x)$ & $(\forall\, i : 1 \le i < k : A[i] \ne x)$.
     ▯ $A[k] = x \xrightarrow{\text{exit}}$
     ▯ $A[k] \ne x \longrightarrow k := k + 1$
    od
    Postconditions: $A[k] = x$ & $(\forall\, i : 1 \le i < k : A[i] \ne x)$.
```

End of linear search

In the problem above, we have assumed that the integer we searched for is in the array. How do we solve the problem when we have no such prior knowledge? One approach is to maintain a boolean flag *present* that tells us whether or not the search was successful.

Linear search with a flag

```
Nat n;
Int A[1..n];
Int x;
Boolean present;
    Preconditions: $0 < n$.
Nat k;
    $k, present := 1, A[1] = x$;
    do
        Variant: $n - k$.
        Invariant: ($present$ & $A[k] = x$ & $(\forall\, i : 1 \le i < k : A[i] \ne x)$)
                                      or (not $present$ & $(\forall\, i : 1 \le i \le k : A[i] \ne x)$).
     ▯ $present \xrightarrow{\text{exit}}$
     ▯ not $present$ & $k < n \longrightarrow k := k + 1$;
                                  if
                                   ▯ $A[k] = x \longrightarrow present := true$
                                   ▯ $A[k] \ne x \longrightarrow$ skip
                                  fi
    ▯ $k = n \xrightarrow{\textbf{exit}}$
    od

    Postconditions: $(\textit{present}\ \&\ A[k] = x\ \&\ (\forall i : 1 \le i < k : A[i] \neq x))$
                         or $(\textbf{not}\ \textit{present}\ \&\ (\forall i : 1 \le i \le n : A[i] \neq x))$.
```

End of linear search with a flag

Actually, the above program is not liked by anybody. One can think of making it better by small adjustments. However, serious improvement is achieved by a conceptual breakthrough. Note the simplicity of the solution when we knew that the element we searched for was in the array. At very little cost we can get the simplicity even when we do not know that. We will simply insert the searched for element into the array, at an additional entry, added to the array.

Linear search with a sentinel

```
Nat $n$;
Int $A[1..n+1]$;
Int $x$;
Boolean $\textit{present}$;

    $A[n+1] := x$;

    Preconditions: $\exists i : 1 \le i \le n+1 : A[i] = x$.
Nat $k$;
    $k := 1$;
    do
        Variant: $n + 1 - k$.
        Invariant: $(\exists i : k \le i \le n+1 : A[i] = x)\ \&\ (\forall i : 1 \le i < k : A[i] \neq x)$.

    ▯ $A[k] = x \xrightarrow{\textbf{exit}}$
    ▯ $A[k] \neq x \longrightarrow k := k + 1$
    od
    $\textit{present} := k \neq n + 1$

    Postconditions: $(\textit{present}\ \&\ A[k] = x\ \&\ (\forall i : 1 \le i < k : A[i] \neq x))$
                         or $(\textbf{not}\ \textit{present}\ \&\ (\forall i : 1 \le i \le n : A[i] \neq x))$.
```

End of linear search with a sentinel

The search process can be substantially more efficient if we know something more about the array A, for instance when we know that A is sorted in non-decreasing order. Then we can apply the so called binary search.

Binary search

```
    $Nat\ \ n$;
    $Int\ \ A[1..n]$;
    $Int\ \ x$;
    $Boolean\ \ present$;
        Preconditions: $\forall\, i : 1 \le i < n : A[i] \le A[i+1]$.
    $Nat\ \ k, l, r$;
        $l, r, present := 1, n+1, false$;
        do
            Variant: $r - l - ord(present)$.
            Invariant: ($present$ implies $A[k] = x$)
                    & (not $present$ implies $(\forall\, i : 1 \le i < l$ or $r \le i \le n : A[i] \ne x)$).
          ▯ $present \xrightarrow{\textbf{exit}}$
          ▯ not $present$ & $l < r \longrightarrow k := (l + r)$ div $2$;
                                      if
                                        ▯ $A[k] = x \longrightarrow present := true$
                                        ▯ $A[k] > x \longrightarrow r := k$
                                        ▯ $A[k] < x \longrightarrow l := k + 1$
                                      fi
          ▯ $l = r \xrightarrow{\textbf{exit}}$
        od
        Postconditions: ($present$ implies $A[k] = x$)
                                & (not $present$ implies $(\forall\, i : 1 \le i \le n : A[i] \ne x)$).
```

End of binary search

Note: in the first disjunct of the invariant and in the first disjunct of postconditions we make reference to k which may be undefined in the case when the array is empty. Is this allowed? Note that *present* is false in this case.

Note: binary search does not guarantee that if x occurs in A, then k has the smallest possible value.

1.11 Selection sort

Point: Start with knowing nothing and accumulate your knowledge by doing something.

Point: How to reason about nested loops?

Given a non-empty array $A[1..s]$ of rational numbers. Sort the array in non-decreasing order.

Selection Sort

```
Nat  $s$;
Rat  $A[1..s]$;
     **Preconditions**: $s > 0$.
Nat  $i, j, a$;
     $i := 1$;
     **do**
          **Variant**: $s - i$.
          **Invariant**:
               $\forall k : 1 \leq k < i : A[k] \leq A[k+1]$ & $(\forall l : i \leq l \leq s : A[k] \leq A[l])$.
       ▯ $i = s \xrightarrow{\text{exit}}$
       ▯ $i < s \longrightarrow$  $a, j := i, i+1$;
                        **do**
                             **Variant**: $s - j + 1$.
                             **Invariant**: $\forall k : i \leq k < j : A[a] \leq A[k]$.
                          ▯ $j = s + 1 \xrightarrow{\text{exit}}$
                          ▯ $j \leq s \longrightarrow$  **if**
                                              ▯ $A[a] \leq A[j] \longrightarrow$ **skip**
                                              ▯ $A[a] > A[j] \longrightarrow a := j$
                                           **fi**
                                           $j := j + 1$
                        **od**
                        **Conditions**: $\forall k : i \leq k \leq s : A[a] \leq A[k]$.
                        $A[i], A[a] := A[a], A[i]$;
                        $i := i + 1$
     **od**
**Postconditions**: $\forall i : 1 \leq i < s : A[i] \leq A[i+1]$.
```

Selection Sort

1.12 Quotient and remainder

Point: The winding road from specification to program.

The problem definition (version 0): Given two integer values x and y, find integer values q (for $\underline{q}$*uotient*) and r (for $\underline{r}$*emainder*) that satisfy the equation:

$$x = q \cdot y + r.$$

A merry hacker may start typing in some code without even noticing that the problem is ill defined. Namely, given any integer values x and y, the above equation admits numerous solutions for q and r. (Try to find different solutions for $x = 7$ and $y = 4$.) Which solution are we interested in? Let us impose some restrictions on r.

The problem definition (version 1): Given two integer values x and y, find integer values q and r such that $0 \le r < y$ and satisfying the following equation:

$$x = q \cdot y + r.$$

The situation seems to be much better. But the problem is still ill defined. Namely, when $y \le 0$, no r satisfies $0 \le r < y$. If we impose a restriction on y that excludes the above possibility then we obtain:

The problem definition (version 2): Given two integer values x and y, $0 < y$, find integer values q and r satisfying the following conditions:

$$\begin{aligned} &0 \le r < y, \\ &x = q \cdot y + r. \end{aligned}$$

Exercise: Check that for any x and y satisfying the preconditions, there is unique q and r satisfying the postconditions. How do you start?

The above is the specification of the Pascal **mod** operation as given in the language standard: r is the value returned by x **mod** y. Note however, that q does not specify the value returned by Pascal x **div** y. Needless to say, not all Pascal compilers satisfy the specification. There is a good reason that the function **mod** in Pascal is not named **rem**. Both these functions mean the same only for nonnegative values of x. Intuitively, a remainder is what remains of the dividend (our x) after the divisor (our y) has been divided into it. In this intuitive sense, a non-zero remainder should have the same sign as the dividend and its absolute value must be smaller than the absolute value of the divisor. Because of this last restriction the divisor must not be zero.

Definition: Function $sign : Int \mapsto \{-1, 0, 1\}$ is defined for integer x as:

$$sign(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ +1 & \text{if } x > 0 \end{cases}$$

Definition: Function $abs : Int \mapsto Int^+ \cup \{0\}$ is defined for integer x as:

$$abs(x) = \begin{cases} -x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

The problem definition (version 3): Given two integer values x and y, $y \neq 0$, find integer values q and r satisfying the following equation:

$$\begin{aligned} & r \neq 0 \ \textbf{implies} \ sign(r) = sign(x), \\ & 0 \leq abs(r) < abs(y), \\ & x = q \cdot y + r. \end{aligned}$$

This statement can be directly turned into a program specification:

The q/r specification

```
Int  $x, y, q, r$;
     Preconditions: $y \neq 0$;

     $\boxed{?}$

     Postconditions: $r \neq 0$ implies $sign(r) = sign(x)$,
                     $0 \leq abs(r) < abs(y)$,
                     $x = q \cdot y + r$.
```

End of the q/r specification

Exercise: Check that the specification is meaningful, that is, for every pair of input values x and y satisfying the preconditions, there is a unique pair of output values q and r satisfying the postconditions. *Only when the check is successful can we think about developing a program for the specification.*

Let us develop the program assuming that we do not have a function **div** available to us. If such a function were available to us, we would have an immediate solution.

Since the postconditions are expressed in terms of *sign* and *abs* functions, one may suspect that we will have to consider 4 cases depending on the signs of the input values. Fortunately, the task is much easier if we separately compute the sign and the absolute values of the outputs.

How can we find $abs(q)$ and $abs(r)$? From the last postcondition we have that $x - r = q \cdot y$. Note that $abs(q \cdot y) = abs(q) \cdot abs(y)$. Now, when $r = 0$ then certainly $abs(x) = abs(q) \cdot abs(y) + abs(r)$. When $r \neq 0$, then $sign(x) = sign(r)$, and therefore $abs(x - r) = abs(x) - abs(r)$, thus $abs(x) = abs(q) \cdot abs(y) + abs(r)$. Note that this implies $abs(x) \geq abs(r)$.

What is $sign(q)$? Look at the last condition again as $x - r = q \cdot y$. Note that $sign(x - r) = 0$ only when $x = r$. Otherwise, when $x \neq r$ we have $sign(x - r) = sign(x)$ because of the first postcondition and the last observation from the previous paragraph. This means that when $q \neq 0$ then $sign(x) = sign(q) \cdot sign(y)$ and therefore $sign(q) = sign(x) \cdot sign(y)$.

To summarize, we know how to find the sign of q and r from just knowing whether they are *0* or not. With this observation we design the program to compute the $abs(q)$ and $abs(r)$ first and then we will adjust the signs.

The invariant of the loop is derived from the postconditions.

The q/r specification

```
Int  $x, y, q, r$;
     **Preconditions**: $y \neq 0$;
Nat  $ax, ay := abs(x), abs(y)$;
     $q, r := 0, ax$;
     **do**
          **Variant**: $r$;
          **Invariant**: $ax = q \cdot ay + r$;
     $\boxed{?}$
     **od**
     **Loop postconditions**: $0 \leq r < ay \ \& \ ax = q \cdot ay + r$
     $\boxed{?}$
     **Postconditions**: $r \neq 0$ **implies** $sign(r) = sign(x)$,
                         $0 \leq abs(r) < abs(y)$,
                         $x = q \cdot y + r$.
```

End of the q/r specification

Complete the program. The guard for $\xrightarrow{\textbf{exit}}$ and the loop invariant must imply the loop postconditions. We hope the commands that turn the loop postconditions into the original postconditions are easy to find.

Variation 1: Imagine you have the operation of division of integers with rational result available. How would you write the q/r program? To answer this question you may need the definitions of the *floor*, *ceiling*, and *truncate* functions (function *round* is probably useless). Here are the definitions:

Definition: Let x be a rational and z be an integer. Function *floor* : *Rat* $\mapsto$ *Int* is defined as follows:

$$floor(x) = z \text{ means } z \leq x \ \& \ x < z + 1$$

Definition: Let x be a rational and z be an integer. Function *ceil* : *Rat* $\mapsto$ *Int* is defined as follows (*ceil* for *ceiling*):

$$ceil(x) = z \text{ means } z - 1 < x \ \& \ x \leq z$$

Definition: Let x be a rational and z be an integer. Function *trunc* : *Rat* $\mapsto$ *Int* is defined as follows:

$$trunc(x) = \begin{cases} ceil(x) & \text{if } x \leq 0 \\ floor(x) & \text{if } x > 0 \end{cases}$$

Definition: Let x be a rational and z be an integer. Function *round* : *Rat* $\mapsto$ *Int* is defined as follows:

$$round(x) = \begin{cases} ceil(x - 0.5) & \text{if } x \leq 0 \\ floor(x + 0.5) & \text{if } x > 0 \end{cases}$$

Variation 2: What are the consequences of dropping the first conjunct of the postconditions?

Variation 3: How can you make the q/r computation faster without using **div**?

1.13 Euclid's algorithm

Note: For historical reasons this should be the first algorithm presented to anybody.

Definition: Given two integer numbers x and y, we say that x divides y, $x \mid y$ for short, if there exists an integer z such that $z \cdot x = y$.

In a slightly different notation, the definition can be stated as follows:

Definition: Given $x, y \in \mathit{Int}$. $x \mid y$ means $(\exists\, z \in \mathit{Int} :: z \cdot x = y)$.

Definition: Given two natural numbers x and y, at least one of them non-zero, we define the *greatest common divisor* of x and y, $gcd(x, y)$ for short, as the largest natural number that divides both x and y.

In a slightly different notation the definition can be stated as follows:

Definition: Given $x, y \in \mathit{Nat}$ such that $\{x, y\} \neq \{0\}$. Let $z \in \mathit{Nat}$. $gcd(x, y) = z$ means

$$z \mid x \;\;\&\;\; z \mid y \;\;\&\;\; (\forall\, u : u \mid x \;\;\&\;\; u \mid y : u \leq z).$$

The following 'recipe' for computing the greatest common divisor of two numbers is attributed to Euclid (Greece, ca. 300 B.C.).

Euclid's recipe

```
Nat  $X, Y$;
     $read(X, Y)$;
Nat  $x, y$;
     $x, y := X, Y$;
     do
       [] $x = y \xrightarrow{\textbf{exit}}$
       [] $x > y \longrightarrow x := x - y$
       [] $x < y \longrightarrow y := y - x$
     od

     $print\ (\ x\ )$
```

End of Euclid's recipe

What you see above **is not** a program. At best, it is an opinion. This opinion is a puzzle, as long as we do not positively answer the following two questions:

- Does Euclid's recipe stop? Why?
- Is the output value equal to $gcd(X, Y)$? Why?

In trying to answer these questions we will have to appeal to some properties of *gcd*. Here is a list of some of them:

(a) $gcd(x, y) = gcd(y, x)$,

(b) $gcd(x, y) = gcd(x + y, y)$,
(c) $gcd(x, y) = gcd(x - y, y)$,
(d) $gcd(x, y) = gcd(x \cdot y, y)$, when $y \neq 0$
(e) $gcd(x, x) = x$,
(f) $gcd(x, 0) = x$,
(g) $gcd(q \cdot x, x) = x$.

Whenever we design a program we better know what the program is supposed to do. The statement of what the program is supposed to do is called the program **specification**. Each program specification defines 3 pieces:

- **Material** that is manipulated (input and output data).
- **Preconditions**, which is the statement of the restrictions on the input data.
- **Postconditions**, which is the statement of the required properties of the output data.

We should have started with the specification of the Euclid's algorithm before writing any code. Better late than never, here it is:

Specification of Euclid's algorithm

Nat X, Y, r;
Preconditions: $X > 0$ & $Y > 0$.

Postconditions: $r = gcd(X, Y)$.

End of the specification of Euclid's algorithm

Note that we do not mention the *read* and *print* commands: the essence of this algorithm is independent of the way the input parameters are taken and the output parameters are reported. For this reason we will not mention the input/output commands in our solution.

We give our reasons why the program presented below is correct wrt (read: with respect to) the specification.

1. That the loop terminates is guaranteed by the expression named **Variant**. This expression is strictly positive and its value decreases every time the loop is executed. Therefore it must stop decreasing at certain point, which means that the loop terminates.

   (*Remark.* The above is equivalent to referring to the *least element principle* for natural numbers, which we will look into when we study induction.)

2. That the postconditions are satisfied is demonstrated with the help of the statement labelled **Invariant**. Note that:

   (a) Before the loop is executed for the first time, the invariant holds.

   (b) If the invariant is true and the loop is executed once more, the invariant will still be true.

   (c) Therefore, the invariant is always true.

(*Remark.* The above reasoning is an example of inductive reasoning over natural numbers, which we will study at length.)

3. Since the invariant is always true, it must be then true when the loop terminates (that it does terminate we know from 1). When the loop terminates we have that $x = y$, and we have the invariant $gcd(x, y) = gcd(X, Y)$. The properties of gcd tell us that under such conditions $gcd(x, y) = x$. Thus, after the $r := x$ assignment is executed, the postconditions are clearly true.

The way we gave our justification is far from formal. However, without further ado we claim that it is *easy to see*[3] that the program has the claimed properties.

Euclid's algorithm

```
Nat  $X, Y, r$;
     Preconditions: $X > 0 \,\&\, Y > 0$;
Nat  $x, y$;
     $x, y := X, Y$;
     do
         Variant: $x + y$.
         Invariant: $gcd(x, y) = gcd(X, Y)$.

       [] $x = y \xrightarrow{\textbf{exit}}$
       [] $x > y \longrightarrow x := x - y$
       [] $x < y \longrightarrow y := y - x$
     od
     $r := x$

     Postconditions: $r = gcd(X, Y)$.
```

End of Euclid's algorithm

[3]Whenever they write that *it is easy to see that ...* it usually means that they do not know how to explain it any better (like in the case above), or they simply do not know how to explain it at all, or that the claimed thing is simply false. The poor reader is to figure out which is the case. Reading—if you insist on understanding what you read—of mathematical texts (or any other meaningful thing) is an activity that can make you sweat.

Variation on Euclid's algorithm

Definition: Given two non-zero natural numbers x and y, the *least common multiple* of x and y, $lcm(x, y)$ for short, is the smallest non-zero natural number that is divisible by both x and y.

In a slightly different notation the definition can be stated as follows:

Definition: Given $x, y \in Nat \setminus \{0\}$. Let $z \in Nat$. $lcm(x, y) = z$ means

$$x \mid z \ \& \ y \mid z \ \& \ (\forall u : x \mid u \ \& \ y \mid u : z \leq u).$$

With the above definitions in mind, consider the following program:

Euclid's algorithm variation 1

```
Nat  $X, Y, r, l$;
     **Preconditions**: $X > 0 \,\&\, Y > 0$;
Nat  $x, y, u, v$;
     $x, y, u, v := X, Y, Y, X$;
     **do**
          **Variant**: $x + y$.
          **Invariant**: $gcd(x, y) = gcd(X, Y) \ \& \ xu + yv = 2XY$.
       ▯ $x = y \xrightarrow{\textbf{exit}}$
       ▯ $x > y \longrightarrow x, v := x - y, v + u$
       ▯ $x < y \longrightarrow y, u := y - x, u + v$
     **od**
     $r, l := x, (u + v)$ **div** $2$
     **Postconditions**: $r = gcd(X, Y) \ \& \ l = lcm(X, Y)$.
```

End of Euclid's algorithm variation 1

That the program terminates is clear from the original Euclid's algorithm. Using the stated invariant and (the not stated here) properties of *gcd* and *lcm*, prove that the above program is correct wrt the specification.

Extended Euclid's algorithm

So far we have been computing the *gcd* using only subtraction. Here we allow ourselves to use division and we also solve a more interesting problem. Try to convince yourself that the program below is correct.

Extended Euclid's algorithm

```
Nat  $X, Y, r$;
Int  $a, b, a', b'$;
         Preconditions: $X > 0 \;\&\; Y > 0$.
Nat  $x, y$;
         $a, b, a', b' := 0, 1, 1, 0$;
         if
           $[\!]\ X \geq Y \longrightarrow x, y := X, Y$
           $[\!]\ X < Y \longrightarrow x, y := Y, X$;
                              $a, b, a', b' := a', b', a, b$
         fi
         Conditions: $\{x, y\} = \{X, Y\} \;\&\; 0 \leq y \leq x \;\&$
                     $a \cdot X + b \cdot Y = y \;\&\; a' \cdot X + b' \cdot Y = x$.
         do
              Variant: $y$.
              Invariant: $gcd(x, y) = gcd(X, Y) \;\&$
                         $0 \leq y \leq x \;\&$
                         $a \cdot X + b \cdot Y = y \;\&\; a' \cdot X + b' \cdot Y = x$.

           $[\!]\ y = 0 \xrightarrow{\textbf{exit}}$
           $[\!]\ y \neq 0 \longrightarrow a, b, a', b' := a' - (x \textbf{ div } y) \cdot a, b' - (x \textbf{ div } y) \cdot b, a, b$
                         $x, y := y, x \textbf{ rem } y$;
         od
         $r := x$

         Postconditions: $r = gcd(X, Y) \;\&\; a \cdot X + b \cdot Y = 0 \;\&\; a' \cdot X + b' \cdot Y = r$.
```

End of extended Euclid's algorithm

Problem: Replace the question marks below to make the program correct.

Extended slow Euclid's algorithm

```
Nat $X, Y, r, a, b$;
    Preconditions: $X > 0 \mathbin{\&} Y > 0$.
Nat $x, y, a', b'$;
    $a, b, a', b', x, y := 1, 0, 0, 1, X, Y$;
    do
        Variant: $x + y$.
        Invariant: $gcd(x, y) = gcd(X, Y) \;\&$
                   $a \cdot X + b \cdot Y = x \;\&\; a' \cdot X + b' \cdot Y = y$.
      ▯ $x = y \xrightarrow{\textbf{exit}}$
      ▯ $x > y \longrightarrow x, a, b := x - y, ?, ?$
      ▯ $x < y \longrightarrow y, a, b := y - x, ?, ?$
    od
    $r := x$
    Postconditions: $r = gcd(X, Y) \;\&\; a \cdot X + b \cdot Y = r$.
```

End of extended slow Euclid's algorithm

1.14 The 2, 3 problem

Point: Innocent looking things can be intractable.

The following problem seems to be very hard. The solution is unknown as of this writing. Can you find a variant for the loop in the code below? If you can, tell us immediately.

The 2, 3 problem

```
Nat $x$;
        Preconditions: $x > 0$.
        do
                Variant: ?
                Invariant: $x > 0$.
           ▯ $x = 1 \xrightarrow{\textbf{exit}}$
           ▯ $odd(x) \longrightarrow x := 3 \cdot x + 1$
           ▯ $even(x) \longrightarrow x := x \textbf{ div } 2$
        od
        Postconditions: $x = 1$.
```

End of the 2, 3 problem

Part II
Logic

2 The Island of Knights and Knaves

We begin our study of logic with a collection of puzzles popularized by the logician Raymond M. Smullyan in his book *What Is the Name of This Book? — The Riddle of Dracula and Other Logical Puzzles* (Prentice-Hall, 1978).

There is an island in which every inhabitant is either a knight or a knave. Knights always tell the truth, while knaves always lie. As a visitor, you came upon two inhabitants which we will call A and B. Person A says "I am a knave or B is a knight.". Can you determine what A and B are?

Argument:

First of all, we get rid of the self reference, "I" in the statement made by A and write an equivalent statement:

"A is a knave or B is a knight."

which we will call S. Thus A says statement S.

Now, by assumption, A is either a knight or a knave. Let's assume that A is a knave and see what happens.

Thus in saying "I am a knave or B is a knight." A has told the truth. But this is impossible because we have assumed that A is a knave and so always lies. Therefore statement S must be false. That is, A cannot be a knave, and B cannot be a knight.

So assuming that A is a knave leads to an absurd situation. Thus A must not be a knave, and, since there are only knights and knaves, we conclude that A must be a knight.

Now this means that A is telling the truth and so statement S must be true. The first part "A is a knave" is clearly false. This means that the second part "B is a knight" must be true.

Thus, both A and B are knights.

Now most people who solve these kinds of puzzles for recreation would agree that this is a valid argument, that is, it does not make any logical errors in its reasoning, and comes to a correct conclusion. They would also agree that there is more to this argument than just the words above — there is also a large collection of unstated assumptions and rules involved. Here are just a few that are used in the argument above.

- An argument involves objects or things (e.g. island inhabitants) which may have names (e.g. A, B). In this particular case, different names name different things. But this is not required, some things may have many names.
- Things have properties (e.g. being a knight). For a given property, each thing either has it or not. Things cannot simultaneously have and not have a property.
- An argument contains statements that can be true or false, depending on the context in which the statements are made.
- It is absurd for a statement to be both true and false in the same context.

- An argument begins by making some statements that set the context is which the argument is held. These statements are assumed to be true. Then, each step in the argument follows from the previous steps by employing some rule of inference.
- Each correct step results in a new true statement, that is, everything we state in an argument is supposed to be true in the context in which it is stated.
- Arguments may contain sub-arguments.
- Simple statements can be composed into complicated statements by using connectives like *and*, *or*, and *not*. The meaning of the complicated statement depends on the the meaning of its constituent simple statements and the connectives used to construct the statement.
- Statements can be manipulated in ways that preserve their meaning but change their form.

Our task will be to clarify these and other properties of an argument. In the process we will be introducing a *formal logical system* called MIZAR MSE. Developing this formal system requires us to address issues of both *syntax* and *semantics*.

Syntax refers to the form of the sentences that we construct, while semantics refers to the meaning that we assign to the sentences. In general, issues of syntax can be addressed without introducing semantics. For example, one can determine that an English sentence is grammatically correct without actually understanding what it says. You simply check that the sentence does not break any rules of the grammar. In describing the syntax of MIZAR sentences and proofs we will use a special kind of grammar called Backus-Naur Form, or BNF for short.

Once one has a precise syntax, it is possible to address the issue of semantics — that is, how do we give meaning to the sentences in the formal system. Usually there are many possible semantics for a given system, and so we must always be clear about which particular one is being used.

In general, we can associate two levels of semantics with a system. One level is fixed and unchanging. It deals with the meaning that we assign to connectives like *and* and *or*, and what it means for something to be a predicate or term. We call this the *fixed semantics*. The other level of semantics is variable and subject to differing *interpretations*. This variable part of the semantics specifies the kinds of things we want to talk about, and defines the meaning of each particular term and predicate used.

To give you some feel for what a formal argument in MIZAR looks like, here is a formal version of the argument above. (kandk01.mse)

```
environ

     given A, B being PERSON;

Opp: for a being PERSON holds Knight[a] iff not Knave[a];

    == These two statements form a translation of the sentence, said by A:
    == "I am a knave or B is a knight"

A1:  Knight[A] implies ( Knave[A] or Knight[B] );
A1': Knave[A] implies not ( Knave[A] or Knight[B] );

begin

    == The question is what are A and B?

    == First we argue that A is a knight.

C1: Knight[A]
    proof
        assume 0: not Knight[A];
               1: Knave[A] by 0, Opp;
               2: not( Knave[A] or Knight[B]) by 1, A1';
               3: not Knave[A] by 2;
        thus contradiction by 1, 3;
    end;

    == Now we argue what B is.

S1: Knave[A] or Knight[B] by C1, A1;
S2: not Knave[A] by C1, Opp;
    Knight[B] by S1, S2;
```

3 The Syntax and Fixed Semantics of MIZAR MSE

These next sections define the syntax and fixed semantics of the MIZAR Multisorted Predicate Calculus with Equality, abbreviated MIZAR MSE.

3.1 Identifiers, Sorts, Constants, Variables, and Terms

One of the first issues that we must grapple with is the problem of identifying the objects that we wish to talk about. This can be done either directly or indirectly.

The direct way of identifying an object is to give it a name or an *object identifier*. The *value* or *denotation* of an object identifier is the object currently named by the identifier. The precise syntactic specification of an object identifier is given by the following BNF grammar (for the notation see section B.1):

ObjectId ::= *Identifier*

Identifier ::= { *IdentifierChar* }$^+$ { `'` }

IdentifierChar ::= `A..Z` | `a..z` | `0..9` | `_`

Thus `0`, `0''`, `1a2`, `aa'`, `Hello` and `Good_bye` are all syntactically correct object identifiers.

Instead of simply having objects of only one kind, we divide the *universe of discourse* into disjoint collections called *sorts*. Sorts are given names, just like objects.

SortId ::= *Identifier*

We often use only capital letters to name sorts. So typically a sort will have an identifier like `THING`, or `OBJECT`. But we will also have sorts with names like `Nat` or `Int`.

Sometimes a particular name always refers to one specific object — as in the figure “7” that names the number seven. In this case we say that the identifier “7” is a *constant* (the identifier “7” has constant denotation). But often, as in real life, the same name will refer to different objects depending on the context in which it is used — such as the name Fred. In this case we say the the identifier “Fred” is a *variable* (the denotation of “Fred” is variable). The use of the word “variable” is not meant to imply that the value of the object identifier is fuzzy or undefined, only that it can change depending on context. The relationship between identifiers and objects in a logical system is both analogous to and different from the relationships between identifiers and objects in programming languages.

Some constant declarations are meant to apply to the entire proof. These are called global constants, and have the syntax:

GlobalConstantsDeclaration ::= `given` *QualifiedObjects*

For example the beginning of this MIZAR proof text defines seven constants, three of sort `INTEGER` and four of sort `PERSON`. Such a declaration occurs only in the environment portion of an argument. Unless these object identifiers are redeclared, their denotations are fixed for the remainder of the proof. (ex02.mse)

```
environ
    given 1,2,3 being INTEGER;
    given Fred, Barney, Wilma, Pebbles being PERSON;
```

We can also refer to objects in the *universe of discourse* by the indirect means of expressions involving other directly identified objects. For example, the object identifier `a` directly refers to the object named `a`, while `1+a` is an expression that could refer to the integer that is the successor of the integer named by `a`. Expressions, including just an object identifier by itself, that result in references to objects are called *terms*.

Each *term* is an expression that refers to exactly one object. The MIZAR MSE logical system supports only simple terms syntactically consisting of an object identifier.

Term ::= *ObjectId*

This lets us ignore the various issues associated with expressions, such as the meaning of the `+` operator on objects that are not numbers. Thus the precise syntactic specification of a term is rather simple: Each term refers to a specific object of a certain sort, and in any given context, a term can belong to only one sort.

3.2 Predicates and Atomic Formulas

The most basic kind of statement that we can make about a thing is that some fact does or does not hold for it. For example suppose that A and B are two constants of some, unspecified, sort:

"It is raining."

"A is a knight."

"A is taller than B."

we can read these as, respectively,

The fact "it is raining" is true.

The "knight" characteristic holds for A.

The "taller than" relationship holds between A and B.

Observe that these statements are about things — 0,1, or 2 things in this case. Each of these statements is an example of a *predicate*. Each predicate has 0, 1, or more arguments, and the meaning of the predicate depends on the things named by the arguments. The predicate is *true* when the property holds among the given arguments, and it is *false* when the property doesn't hold.

For example we might interpret the predicates `Raining`, `Knight`, and `Taller` in the obvious way, and then apply them to the objects `A` and `B` to obtain three sentences

```
Raining[]

Knight[A]

Taller[A, B]
```

which correspond in meaning to the three previous ones. Note how the `Raining` predicate has no arguments, and so is applied to the empty list `[]`.

The number of arguments that a predicate has is called its *arity.* We have special names for predicates with certain arities. A predicate with one argument is a *property.* When an object is fixed in some context, the object either has the property or does not. Properties are generally used to make statements about individuals.

A predicate with 2 or more arguments is a *relation*. Given a number of objects in a context, and given a specific relation, the relation either holds among the objects or it does not. Relations are generally used to describe relationships among things.

We do not give a special name to a predicate with no arguments. It is true or false, depending on the context in which it appears. Such a predicate is generally used as an abbreviation for a more complex statement about the world that we prefer not to repeatedly write in long form.

The precise specification of the meaning of a predicate depends in general on the specific *interpretation* being used. For example `less[A,B]` could mean that object `A` is smaller than object `B`, or that object `B` is smaller than object `A`. Assigning meaning to the predicates is accomplished through the interpretation.

However, we have two special predicates whose meaning never changes. They are called *equality* and *inequality*. For example `A=B` means `A` and `B` name the same thing, while the formula `A<>C` means that `A` and `C` name different things.

All of these simple statements are called *atomic formulas*. Use of the term "formula" indicates that we are concerned with the form of the statement, not its content. The term "atomic" indicates that these objects are the building blocks for more complex formulas. Here is the BNF grammar for an atomic formula.

AtomicFormula ::= *Equality* |
Inequality |
PredicativeFormula |
`contradiction`

Equality ::= *Term* `=` *Term*

Inequality ::= *Term* `<>` *Term*

PredicativeFormula ::= *PredicateId* `[` [*Term* { `,` *Term* }] `]`

The reserved word `contradiction` is a formula that is always false. Observe that `Knight`, `Raining`, and `Taller` are all legal predicate identifiers, and that `Knight[]`, `Knight[A]`, and `Knight[x,y]` are all syntactically legal atomic formulas.

3.3 Logical Connectives and Compound Formulas

We can combine atomic formulas into more complex sentences using the connectives `&` (read as "and"), `or`, `not` and `implies`. The syntactic rules for constructing sentences from these connectives are given by the following BNF grammar.

Formula ::= *AndOrFormula* |
ConditionalFormula |
BiconditionalFormula |
QuantifiedFormula

AndOrFormula ::= *ConjunctiveFormula* |
DisjunctiveFormula

DisjunctiveFormula ::= *ConjunctiveFormula* { `or` *ConjunctiveFormula* }

ConjunctiveFormula ::= *UnitFormula* { `&` *UnitFormula* }

UnitFormula ::= { `not` } (*AtomicFormula* | `(` *Formula* `)`)

ConditionalFormula ::= *AndOrFormula* `implies` *AndOrFormula*

BiconditionalFormula ::= *AndOrFormula* `iff` *AndOrFormula*

(Quantified formulas are discussed in section 3.5.)

Note that some connectives are treated in different productions with slightly different forms. This is because we assign a different *precedence* to each of the operators in order to save us from writing one pair of parentheses for each connective. The precedence of the operators, in decreasing order is:

`not`
`&`
`or`
`implies`, `iff`

A convenient rule of thumb for remembering the precedence of the operators is to correspond `not` with unary minus, `&` with $\times$, and `or` with $+$ in the usual arithmetic expressions. The logical connectives `&` and `or` are associative just like $\times$ and $+$.

Putting parentheses (`()`) around a subformula has the usual effect of making the enclosed subformula into a single unit with respect to the remainder of the enclosing formula. The operators `implies` and `iff` have equal priority, but they are not associative. This fortunately does not create a problem of ambiguity because the grammar ensures that `implies` and `iff` can appear only between subformulas that do not contain unparenthesized `implies` and `iff` operators. In brief, MIZAR logical formulas are evaluated in the same way that logical expressions in Pascal or C are evaluated.

The grammar tells us how to construct complex sentences from existing ones. We give them meaning by showing how to obtain the truth of the complex sentence from the meaning of its individual parts.

Let ϕ and ψ stand for arbitrary MIZAR sentences, except that we assume that the sentences are enclosed in `()`. This avoids some technical difficulties caused by operators having unequal precedence. (Consider the case of placing an `&` between the formulas `P[] or Q[]` and `R[] or S[]`. If we did not enclose each formula in parentheses, placing an `&` between them would result in the formula `P[] or Q[] & R[] or S[]` instead of our intended one of `(P[] or Q[]) & (R[] or S[])`. Users of macro processors are familiar with this problem.)

The simplest connective is `not`, and `not` ψ is true if ψ is false, and false if ψ is true. We can express this using a *truth-table*.

ϕ	`not` ϕ
F	T
T	F

The remaining connectives take two arguments, and are defined by the following truth-table.

ϕ	ψ	ϕ `&` ψ	ϕ `or` ψ	ϕ `implies` ψ	ϕ `iff` ψ
F	F	F	F	T	T
F	T	F	T	T	F
T	F	F	T	F	F
T	T	T	T	T	T

Note how `&` is true exactly when both propositions are true, and `or` is true when at least one proposition is true. This property lets us avoid having to put parentheses about the subformulas in a conjunctive or disjunctive formula. Thus `(P[] & Q[]) & R[]` is the same as `P[] & (Q[] & R[])`, and `(P[] or Q[]) or R[]` is the same as `P[] or (Q[] or R[])`.

Here are some examples of some simple sentences:

```
not Knight[A] & Knight[B]
```

This can be read as "A is not a knight, and B is a knight". This sentence

```
Knave[A] & (Raining[] or Knight [B])
```

can be read as "A is a knave, and it is raining or B is a knight". Note how we use parentheses to override the basic precedence of the connectives. For example the next sentence has a different meaning from the previous one

```
Knave[A] & Raining[] or Knight[B]
```

but is the same as

```
(Knave[A] & Raining[]) or Knight[B]
```

How do we determine the truth value of a complex sentence? If we know the truth value of the atomic formulas in a complex sentence, then we can determine the truth value of the complex sentence by simply building up successively more complex subsentences — just as we do for arithmetic expressions.

Here are some examples. The sentence

```
A=B & not A=C or A=C & not A=B or B=C & not A=B
```

is true when exactly two of `A`, `B`, `C` are equal. It is false otherwise. Suppose that we have `A=1`, `B=2`, `C=1`. How do we determine the truth value of this formula under this interpretation?

First we break it into its three conjunctive subformulas:

```
A=B & not A=C
A=C & not A=B
B=C & not A=B
```

Since the subformulas are connected together by `or` to form the main formula, the main formula will be true if at least one of its subformulas is true. Next we break each conjunctive formula into its unit formulas. For the first subformula this is

```
  A=B
not A=C
```

This subformula will be true if both of its sub-subformulas are true. From the values of A, B, C given above we see that A=B is false; and A=C is true, so not A=C is false. Either one of the sub-subformulas being false is enough to make the subformula false. Thus far we cannot make any conclusions about the truth value of the main formula.

However, the second conjunctive subformula A=C & not A=B is true, and this makes the main formula true. Thus we have that the main formula is true under the interpretation A=1, B=2, C=1.

We can also perform this process on the parse tree for the formula, and simply assign a logical value to each node. This process can be represented diagrammatically as follows under the interpretation A=1, B=2, C=1.

```
A=B & not A=C or A=C & not A=B or B=C & not A=B
---       ---    ---       ---    ---       ---
 F         T      T         F      F         F
      -------          -------          -------
         F                T                T
-------------    -------------    -------------
    F                T                F
-----------------------------------------------
                     T
```

For our next example, suppose that we have three predicates Odd, Even, and Divides which we interpret as follows:

Odd[x] – is true exactly when x is an odd integer.

Even[x] – is true exactly when x is an even integer.

Divides[x,y] – is true exactly when x divides into y leaving no remainder.

Under this interpretation we have:

Odd[1] is true, Even[1] is false, Odd[2] is false, Even[2] is true
Divides[2,4] is true, Divides[2,5] is false, Divides[2,1] is false

Now suppose x=3 and y=7. What is its truth value of this sentence?

```
(Odd[x] or Even[x]) & not(Even[x] & Odd[x]) or not Divides[x,y]
 ------    -------        -------   ------         ------------
   T         F               F         T               F
-------------------      ------------------    ----------------
        T                          F             T
                      ---------------------
                        T
-------------------------------------------
                    T
---------------------------------------------------------------
                                             T
```

In the case x=4 and y=8 the sentence is also true.

3.4 Conditionals and Biconditionals

One of the more striking aspects of an argument is the frequent repetition of a phrase "If ... then ...". For example, "If Fred is a herring then Fred is a fish". This kind of sentence is called a *conditional* because it states that a conditional relationship holds between the *antecedent* (following the "if") and the *consequent* (following the "then"). If the antecedent is true then the consequent is also true.

All formulas of the form ϕ `implies` ψ are conditionals. They can also be read as "if ϕ then ψ". Think of the sentence ϕ `implies` ψ in the following way. If ϕ `implies` ψ is true, and we also know that ϕ is true, then we can conclude that ψ is also true. This is a direct consequence of the definition of the implication connective. Here is an example of how conditionals are used in an argument:

Suppose that

`Knight[A]`,
`Knight[A] implies Knave[B]`, and
`Knave[B] implies Knight[C]`

are all true. Then from the first two we can conclude that `Knave[B]` is true.
From the third, since `Knave[B]` is true we can conclude that `Knight[C]` is true.

The `implies` connective can be expressed in terms of `not` and `or`. Specifically, ϕ `implies` ψ is equivalent to `not` ϕ `or` ψ. You can check to see that these two sentences have exactly the same truth-table. Remember this equivalence!

A *biconditional* is the combination of two conditionals. The biconditional ϕ `iff` ψ, read "ϕ if and only if ψ", or "ϕ is equivalent to ψ", simply means `(`ϕ `implies` ψ`) & (`ψ `implies` ϕ`)`.

The ability to transform formulas into equivalent ones is important, and you should make an attempt to become thoroughly familiar with the various techniques that we introduce.

3.5 Quantifiers

In English, we often say things like:

"All mammals have hair."
"Some mammals have four legs."

These sentences state facts, not about some specific individual, but about classes of individuals. We can interpret the first sentence as saying that no matter what mammal you pick, it will always have hair. This is an example of a *universal* statement. The property of having hair applies universally to the class of things we call mammals.

The second sentence can be interpreted as saying that at least one of the members of the class of mammals has four legs. Possibly more, or even all of the members of the class do. This is an example of an *existential* statement, so called, because it asserts the existence of at least one thing satisfying the stated property.

In MIZAR we would translate these two sentences as something like:

```
for x being MAMMAL holds HasHair[x]
 ex x being MAMMAL st FourLegs[x]
```

under the obvious interpretation of the predicates `HasHair` and `FourLegs`.

The phrases `for x being MAMMAL holds` and `ex x being MAMMAL st` are called *quantifications.* The keyword `for` is called the *universal quantifier*, and the keyword `ex` is called the *existential quantifier.* Each quantifier applies to the syntactically complete formula that follows the `holds` or `st`.

In a quantified formula, some object identifiers are associated with a quantifier, while others are not. Those associated with a quantifier are called *bound*, and the relationship between the object identifier and its associated quantifier is called the *binding* of the object identifier. Object identifiers not associated with a quantifier are called *free.* The notions of bound and free occur with respect to a particular formula. An object identifier that is free in one formula may be bound in another formula. (Note that in usual logical parlance the word "variable" is used in a sense similar to the way we use "object identifier".)

3.6 Scope and Binding

It is important to be able to determine the *bindings* of the object identifiers in a formula to the quantifiers in the formula. This is done by determining the *scope* of each quantifier. When a quantifier introduces an object identifier, then every occurrence of the object identifier inside the scope of the quantifier is bound to the quantifier.

The scope of each quantifier is determined according to the following rules.

1. Suppose that ϕ is a formula with no quantifiers. Then every occurrence of every object identifier in ϕ is *free.*

2. Any free object identifiers in ϕ and ψ are still free in, and any bound identifiers retain their bindings in, all of the following formulas

   $$\texttt{not } \phi$$
   $$\phi \texttt{ \& } \psi$$
   $$\phi \texttt{ or } \psi$$
   $$\phi \texttt{ implies } \psi$$
   $$\phi \texttt{ iff } \psi$$

3. Suppose that you have a formula ψ of the form

   $$\texttt{for } \alpha \texttt{ being } \tau \texttt{ holds } \phi$$

   or

   $$\texttt{ex } \alpha \texttt{ being } \tau \texttt{ st } \phi$$

   where α is a object identifier, τ is a sort. Then object identifier α is *bound* in ψ, the scope of the quantifier `for` or `ex` is all of the free occurrences of α in ϕ, and all of (and only) the free occurrences of α in ϕ are bound in ψ to the quantifier `for` or `ex`. All other free occurrences of any object identifier in ϕ are still free in ψ.

We now do a couple of examples. The important thing to remember is that determining the bindings of object identifiers in a logical formula is exactly the same problem as determining the bindings of variable names in a programming language. Consider the following formula:

```
for x,y being THING holds (x=y or P[z] & (ex y being BLOB st B[x,y]))
```

To determine the binding of each of the object identifiers `x,y`, and `z` in this formula we apply the rules we just introduced. We begin by looking at the lowest level component

```
B[x,y]
```

in which `x`, `y` are both free by rule 1. This formula is quantified to make a new statement

```
ex y being BLOB st B[x,y]
```

in which `x` is still free, but `y` is now bound to the `ex` quantifier by rule 3. Now `z` is free in `P[z]`, and so when we apply rule 2 to construct

```
P[z] & (ex y being BLOB st B[x,y])
```

we have all occurrences of `x` and `z` being free, and all occurrences of `y` being bound. Since `x` and `y` are free in `x=y`, when we apply rule 2 to construct

```
x=y or P[z] & (ex y being BLOB st B[x,y])
```

we get that all occurrences of `x` and `z` are free, the first occurrence of `y` is free, and the other occurrences of `y` are bound. Finally, applying rule 3 we get

```
for x,y being THING holds (x=y or P[z] & (ex y being BLOB st B[x,y]))
```

which has all occurrences of `x` and the first occurrence of `y` bound to the `for` universal quantifier; the second occurrence of `y` being bound to the `ex` existential quantifier; and all occurrences of `z` being free.

Here is another example. Formulas 1 and 2 have the same bindings, while 3 does not. (Remember that a quantifier with a list of object identifiers is simply a short form for a sequence of quantifiers, one for each object identifier.)

```
1: for x, y, z being THING holds R[x,y,z]

2: for x being THING holds
     for y being THING holds
       for z being THING holds R[x,y,z]

3: for x, z, y being THING holds R[x, y, z]
```

3.7 The Semantics of Quantified Formulas

What meaning do we give to quantified formulas? Suppose that α is an object identifier, τ is a sort identifier, and ϕ is a formula with free occurrences of variable α. Then a universally quantified formula

$$\texttt{for}\ \alpha\ \texttt{being}\ \tau\ \texttt{holds}\ \phi$$

is true exactly when ϕ is true no matter what object of sort τ is is named by α. If ϕ fails to hold for even one such object, the universal formula is false.

An existentially quantified formula

$$\texttt{ex}\ \alpha\ \texttt{being}\ \tau\ \texttt{st}\ \phi$$

is true exactly when there is at least one object of sort τ that satisfies statement ϕ.

Another way of thinking about the universal and existential quantifiers is as generalizations of the `&` and `or` connectives. Suppose that the objects of some sort `S` could be completely enumerated as the objects `a`, `b`, `c`, `d`. Then saying

```
for x being S holds P[x]
```

is equivalent to the formula

```
P[a] & P[b] & P[c] & P[d]
```

Similarly, saying

```
ex x being S st Q[x]
```

is equivalent to the formula

```
Q[a] or Q[b] or Q[c] or Q[d]
```

We will do some examples later when we look at translations.

3.8 BNF Grammar for Quantified Formulas

Here is the general syntax for a quantified formula. The first rules are used to specify the list of object identifiers being quantified along with their sorts. This list is a shorthand abbreviation of what is to be considered as a number of quantifications, each one applying to only a single identifier. This will be explained when we begin to do inferences using quantified formulas.

QualifiedObjects ::= *QualifiedSegment* { `,` *QualifiedSegement* }

QualifiedSegment ::= *ObjectId* { `,` *ObjectId* } (`be` | `being`) *SortId*

The actual quantified formula is given by

QuantifiedFormula ::= *UniversalFormula* | *ExistentialFormula*

UniversalFormula ::= `for` *QualifiedObjects* `holds` *Formula*

ExistentialFormula ::= `ex` *QualifiedObjects* `st` *Formula*

4 Interpretations

Logical formulas by themselves have no meaning or *truth-value.* To assign meaning to a formula we must specify an *interpretation.* The interpretation must provide sufficient information for us to determine whether the formula is true or false.

Each interpretation must specify two things:

- For each predicate you must identify the sorts of objects that it can be applied to, and specify the value of the predicate on all the relevant arguments.
- For each constant object you must specify its sort, and possibly the actual object that it denotes.

The degree of detail required by an interpretation varies with the formula that you are trying to interpret. For example the formula

```
P[] & (Q[] or R[])
```

only involves predicates of 0 arity. A complete interpretation would require you to only state whether each atomic formula was true or false. But there are circumstances where having only a partial specification of the truth-values of the predicates would still permit you to interpret the formulas. For example if `P[]` is true and `R[]` is true we do not need to know the interpretation of `Q[]` in order to determine that the formula is true. Another case would be where `P[]` was false.

A more complicated situation occurs when the predicates have arguments, but the arguments are all constants. For example

```
Knight[Fred] iff not Knave[Fred]
```

To interpret this formula we need to know what the sort of object `Fred` is, and whether it is of a sort required by the predicates `Knight` and `Knave`. We also need to know whether `Knight[Fred]` and `Knave[Fred]` are true or false. This could be specified explicitly, for example by saying `Knight[Fred]` is true, or implicitly, for example by saying that the `Knight` predicate is always true.

The more common situation is the case where you have a quantified formula such as

```
for x being PERSON holds
Likes[Fred,x] & ( ex y being PERSON st not Likes[y,Fred] )
```

One possible interpretation could be that the `Likes` predicate is always applied to arguments of sort `PERSON`, and that `Likes[a,b]` is intended to mean that person `a` likes person `b`. In addition, `Fred` likes everyone, but `Wilma` does not like `Fred`. Under this interpretation the formula is true.

Note how the generality of the interpretation increases as the formulas we want to interpret become more complex.

4.1 Satisfiability and Falsifiability

Consider these formulas in which all the predicates are of 0 arity.

1. `(P[] & Q[]) or R[]`
2. `(P[] & Q[]) or not (P[] & Q[])`

Under this interpretation

P[]	Q[]	R[]
T	F	F

Formula (1) is false and formula (2) is true.

An interpretation that makes a formula ϕ true is called a *satisfying interpretation* for ϕ. An interpretation that makes a formula ϕ false is called a *falsifying interpretation.* Some formulas are true no matter what interpretation you use. This is the case for formula (2) above. Such a formula is called a *tautology*. The sentence `P[] or not P[]` is an example of one of the more important tautologies.

Similarly, some formulas are false under any interpretation. Such a formula is called a *contradiction*. The formula `Knight[x] & not Knight[x]` is an example one of the more important contradictions.

Formulas that have both satisfying and falsifying interpretations are called *contingencies*. The simplest contingency is a formula like `P[]`.

Mizar has the built-in ability to reason about formulas in which all predicates are either 0-ary or applied to constants. Thus one way to determine if something is a tautology using Mizar is merely to state it. If Mizar accepts it, then it is a tautology. (If not, it may still be a tautology but Mizar ran out of resources while trying to prove it.) For example, Mizar will accept the following facts without any justification. (ex03.mse)

```
environ
begin        == DeMorgan's Laws

    not ( Phi[] & Psi[] ) iff ( not Phi[] or not Psi[] );

    not ( Phi[] or Psi[] ) iff ( not Phi[] & not Psi[] );

    ( Phi[] & Psi[] ) iff not ( not Phi[] or not Psi[] );

    ( Phi[] or Psi[] ) iff not ( not Phi[] & not Psi[] );
```

This verifies four very useful equivalences, called DeMorgan's Laws, which permit you to change a formula using `&` into a formula using `or` and vice-versa.

How do you use Mizar to determine if a formula is a contradiction? If formula ϕ is always false, then `not` ϕ is always true. Thus, to show that something is a contradiction we simply verify that its negation is a tautology. For example: (ex04.mse)

```
environ
begin
```

```
     == Verify that Phi[] & not Phi[] is a contradiction.

not ( Phi[] & not Phi[]);

     == Verify that Phi[] iff not Phi[] is a contradiction.

not ( Phi[] iff not Phi[] );
```

When a formula is neither a tautology nor a contradiction, how do you go about finding a satisfying or falsifying interpretation? One way, which works well for formulas that are short and contain only 0-ary predicates or predicates applied to constants is to compute the truth-value of the formula under all possible truth assignments to the predicates of the formula. For example, the truth-table for `not P[] or (P[] implies Q[])` is:

P[]	Q[]	not P[]	or	(P[] implies Q[])
F	F	T	T	T
F	T	T	T	T
T	F	F	F	F
T	T	F	T	T

Each line of the table corresponds to one possible interpretation. Thus the formula has three satisfying interpretations and one falsifying.

Since each atomic sentence can be either true or false, the number of possible interpretations for a formula containing n distinct atomic sentences is 2^n. Thus the truth-table method does not work so well when the number of predicates gets much larger than five or so. The size of the table quickly becomes unmanageable, at least for humans.

4.2 Translations

Closely associated with the notion of an interpretation is the notion of a *translation*. A translation is the result of converting an imprecise natural language description of a problem into a specific collection of formulas with their interpretation. This is exactly the same process that occurs when you convert a specification of a problem into a computer program to solve it. Constructing a translation can be a difficult problem requiring much creativity.

4.3 Problems in translation

(trans01.mse)

```
environ == Examples of Translations to Predicate Calculus Formulas
        == We will use the basic sorts
        ==     PERSON    -- an inhabitant of the island.
        ==     STATEMENT -- a sentence in the language of the island.
        ==     LOCATION  -- a place on the island.
        == Some basic constants:

given Fred being PERSON;
given Promise being STATEMENT;
given Bedrock, Edmonton being LOCATION;
```

```
     == The following predicates will be used for any x of sort PERSON, s of
     == sort STATEMENT, and p of sort LOCATION with the intended meaning:
     ==        Knight[x]     means that   x is a knight.
     ==        Knave[x]                   x is a knave.
     ==        Normal[x]                  x is a normal.
     ==        Tells[x,s]                 x says the statement s.
     ==        True[s]                    statement s is true.
     ==        LivesIn[x,p]               x lives in location p.
     == Now we do some example translations:

     == Every person is a knight, knave, or normal.  This is not exclusive,
     == a person could be one or more.

1: for y being PERSON holds Knight[y] or Knave[y] or Normal[y];

     == There is a person called Sam who lives in Edmonton, and the
     == only false thing that Sam says is the statement Promise.

given Sam being PERSON;

2: LivesIn[Sam,Edmonton] &
           (for s being STATEMENT
                  holds Tells[Sam,s] & not True[s] implies s=Promise);

     == If any knight makes a statement then that statement is true.
     == Alternatively, knights always tell the truth.

3: for y being PERSON, s being STATEMENT holds
                  Knight[y] & Tells[y,s] implies True[s];

     == Every normal person tells at least one truth and at least one lie.

4: for x being PERSON holds Normal[x] implies
              (ex s,t being STATEMENT st
                         Tells[x,s] & Tells[x,t] & True[s] & not True[t]);

     == There is at least one statement that is both said by some
     == resident of Edmonton and not said by some resident of Edmonton.

5: ex s being STATEMENT st
       ex y,z being PERSON st LivesIn[y, Edmonton] & LivesIn[z, Edmonton] &
                             Tells[y,s] & not Tells[z,s];

     == Every inhabitant of Bedrock tells the truth.
     == Statements 6 and 7 are two ways of saying this.

6: for x being PERSON, s being STATEMENT holds
            LivesIn[x, Bedrock] & Tells[x,s] implies True[s];

7: for x being PERSON holds LivesIn[x,Bedrock] implies
```

```
                 (for s being STATEMENT holds Tells[x,s] implies True[s]);

    == No inhabitant of Bedrock lies.  This is logically equivalent to both
    == of the preceding statements.

7': not ( ex x being PERSON, s being STATEMENT st
                     LivesIn[x,Bedrock] & Tells[x,s] & not True[s] );

    == Some inhabitants of Edmonton are normal, and some are knights.

8: (ex x being PERSON st LivesIn[x,Edmonton] & Normal[x]) &
   (ex x being PERSON st LivesIn[x,Edmonton] & Knight[x]);


    == If a normal person always tells the truth then they are also a knight.

9: for x being PERSON holds
        Normal[x] &
        (for s being STATEMENT holds Tells[x,s] implies True[s])
                                                       implies Knight[x];
    == Fred tells exactly one lie.

10: ex Lie being STATEMENT st (
       Tells[Fred, Lie] & not True[Lie] &
       ( for Lie2 being STATEMENT holds
              (Tells[Fred,Lie2] & not True[Lie2] implies Lie = Lie2)) );

    == Some normal persons tell the truth all of the time, and some of the time
    == all of the normals tell the truth, but it is not the case that
    == all of the normals tell the truth all of the time.

    == How to translate the phrases "all of the time" and "some of the time".
    == I took this as meaning "every statement they make" and
    == "some statement they make".  This removes "time" from the translation.

    == Another problem is figuring out what "some of the time all of the
    == normals tells the truth" means.  I took this as meaning that
    == at least one sentence was stated truthfully by all normals.

11:   ( ex x being PERSON st (Normal[x] &
               (for s being STATEMENT holds Tells[x,s] implies True[x]) ))
    & ( ex s being STATEMENT st (True[s] &
               (for x being PERSON holds Normal[x] implies Tells[x,s]) ))
    & not ( for x being PERSON, s being STATEMENT holds
                Normal[x] & Tells[x,s] implies True[s] );
```

4.3.1 A question

(trans02q.mse)

```
environ

        given 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 being NATURAL;


== The universe of discourse is the set of natural numbers (NATURAL).
== Translate each of the following statements into syntactically correct
== formulas, using the following scheme of abbreviation.

==      Even[x]       x is an even number
==      Odd[x]       x is an odd number
==      Prime[x]      x is a prime number
==      Divisible[x,y]  x is (evenly) divisible by y

        == Examples:

        == 1: 2 is an even prime.

                1: Even[2] & Prime[2];

        == 2: 6 is not a prime because 3 divides it.

                2: Divisible[6,3] implies not Prime[6];

        == End of examples

==  1: Neither 4 nor 6 is a prime number.

==  2: Neither 8 nor 7 is a prime number.

==  3: If 6 divides 9 then 9 is not a prime.

==  4: 10 is a product of 2 and 5.
                == Watch out, the answer is not
                ==      4: Divisible[10,2] & Divisible[10,5];

==  5: 5 is a prime because it is divisible only by 1 and itself.

==  6: No even number is odd.

==  7: Not all odd numbers are prime.

==  8: There is exactly one even prime.

==  9: No prime is neither even or odd.

== 10: 2 is the only even prime number.
== 11: Every non-prime is divisible by at least one prime.

== 12: There are non-primes that are not divisible by any odd prime.

== 13: Every prime number is not divisible by any number but 1 and itself.

== Final remark. Run your stuff through Mizar. So long as your translations
== appear in the environment section, and they are syntactically correct,
== they will be accepted by Mizar.
```

4.3.2 Its answer

(trans02a.mse)

```
environ
      given 1,2,3,4,5,6,7,8,9,10 being NATURAL;

1: not (Prime[4] or Prime[6]);

2: not (Prime[8] or Prime[7]);

3: Divisible[9,6] implies not Prime[9];

4: Divisible[10,2] & Divisible[10,5] &
      (for X being NATURAL
         holds Divisible[10,X] implies X=1 or X=2 or X=5 or X=10);

5:   Divisible[5,1] & Divisible[5,5] &
     (for X being NATURAL holds Divisible[5,X] implies X=1 or X=5)
   implies Prime[5];

6: for X being NATURAL holds Even[X] implies not Odd[X];

7: not (for X being NATURAL holds Odd[X] implies Prime[X]);

8: ex X being NATURAL st Even[X] & Prime[X] &
             (for Y being NATURAL holds Even[Y] & Prime[Y] implies Y=X);

9: for X being NATURAL holds Prime[X] implies Even[X] or Odd[X];

10: for X being NATURAL holds Even[X] & Prime[X] implies X=2;

11: for X being NATURAL holds not Prime[X]
              implies (ex Y being NATURAL st Prime[Y] & Divisible[X,Y]);

12: ex X being NATURAL st not Prime[X] &
      (for Y being NATURAL
         holds Odd[Y] & Prime[Y] implies not Divisible[X,Y]);

13: for X being NATURAL holds Prime[X] implies
      (for Y being NATURAL
         holds Y<>X & Y<>1 implies not Divisible[X,Y]);
```

5 Arguments

An argument is a systematic attempt to convince someone (possibly yourself) that if certain things are assumed to be true then something else must also be true. The statements we assume to be true are called *assumptions, premises,* or *hypotheses*. The statements that we wish to justify are called *conclusions* or *theses.* In essence, an argument is attempting to establish that the following sentence is true:

"If the assumptions are true then the conclusion must also be true."

An *argument* begins with a number of assumptions. These are sentences that we assume are true. It then proceeds by a sequence of steps. At each step, a new true sentence is established by applying an *inference rule* to one or more of the preceding sentences.

Since every step introduces a true sentence, all sentences in an argument are true — either by assumption or as a result of an *inference* (applying an inference rule). There is one caveat that we must keep in mind — each sentence is true only in the context it was written. Taken out of context the sentence may be false, or more likely, meaningless.

Arguments, by their very nature, start by assuming that the premises are true. If not, and one or more are false, it hardly makes sense to continue the argument. This certainly occurs in real life — how often have you tried to sway the opinion of someone who doesn't even agree with your basic assumptions? But an argument can still be correct, that is its conclusion follows logically from it premises, even if the premises are wrong, and especially so if the premises are self-contradictory. **This is very important to remember.** All a correct argument does is: it expresses a relationship between the premises and the conclusion, it says nothing at all about the truth of the premises.

5.1 The Structure of MIZAR Arguments

Let us begin by looking at a simple argument that formalizes the one we did in our explanation of implies. (ex06.mse)

```
environ
        == This is where the premises of the argument are placed
        == each of these sentences is assumed to be true.

        == First we introduce the names of the objects involved
        == in the argument.  These names are global constants.

    given A, B, C being PERSON;

        == Then we state the logical relationships between the objects.

    P0: Knight[A];
    P1: Knight[A] implies Knave[B];
    P2: Knave[B]  implies Knight[C];

        == Now we start the reasoning part of the argument.
begin

    S1: Knave[B] by P0, P1;
    S2: Knight[C] by S1, P2;
```

The single most important thing to remember about an argument is that every statement is true within the context it is written. No consistent proof system ever permits you to write down a statement that is false in its context.

An argument proceeds in two phases. First you state your premises as a collection of formulas that are assumed to be true. This sets up the initial environment or context for the proof. Thus if we are assuming that Fred is a knight, we might include the axiom

```
A1:  Knight[Fred];
```

in the proof environment. If one of your premises is more naturally stated as a falsehood, you will have to recast it so that it can be stated as a true assumption. For example if Fred is not a knave you might include the axiom:

```
A2:  not Knave[Fred];
```

The next phase of the proof consists of the actual reasoning, broken down into a number of steps. Each step consists of a proposition and its justification. The proposition is an assertion that within the current context some specific logical formula is true. The justification uses facts already established to support the proposition. Implicit in the justification is the use of one or more rules of inference.

The basic structure of a MIZAR proof follows this general pattern. In MIZAR an argument is called an *Article*.

```
Article ::=
    environ
        Environment
    [ begin
        TextProper
    ]
```

The *Environment* contains all of the assumptions and declarations used in the argument that follows in the *TextProper*. (Note that the `begin` *TextProper* part is optional.)

The environment consists of zero or more global constant declarations and *axioms*.

```
Environment ::= {
                    Axiom ; |
                    GlobalConstantsDeclaration ;
                }
```

The *GlobalConstantsDeclaration* introduces names of some objects of specified sorts whose properties are discussed in the rest of text. Some of these properties are stated as axioms. An axiom is a named formula which we assume to be true. Grammatically, an axiom looks like:

```
Axiom ::= LabelId : Formula
```

The *TextProper* consists of one or more *Statements* separated by semicolons.

$$TextProper ::= \{ \; Statement \; ; \; \}^{+}$$

We begin by considering the simplest type of a *Statement*, which is the *CompactStatement* with *SimpleJustification* and has the following shape.

Proposition *SimpleJustification*

where a proposition consists of an optional label followed by a formula:

```
Proposition ::= [ LabelId : ] Formula
```

Every proposition that we write must be justified. The *SimpleJustification* directly appeals to some *inference rule* and has the form

```
SimpleJustification ::= [ by Reference { , Reference } ]

Reference ::= LabelId
```

A simple justification can be empty, in which case its truth is (supposedly) obvious. (The other kind of justification uses a proof. We leave it till later.)

Note that in MIZAR, we need only reference the statements necessary to justify the proposition. The related inference rule is figured out automatically by the MIZAR processor.

(ex08.mse) Consider the following example:

```
environ
    given x, y, z being THING;
    A1: A[x] & B[y];
    A2: B[y] & C[z];
begin
    1: A[x] by A1;
    2: C[z] by A2;
    3: A[x] & C[z] by 1,2;
```

In this proof, we implicitly use two inference rules. One that says if `A[x] & B[y]` is true, then `A[x]` is true. The other says that if `A[x]` is true, and `C[z]` is true, then `A[x] & C[z]` is true.

We can write the first of these inference rules as:

$$\frac{\texttt{A[x] \& B[y]}}{\texttt{A[x], B[y]}}$$

and the second as:

$$\frac{\texttt{A[x], C[z]}}{\texttt{A[x] \& C[z]}}$$

The notation means that if the *all of* the formulae above the line (called *premises*) appear in the proof so far, you can then write in the proof *any* of the statements that appear below the line (called *conclusions*). To illustrate the generality of each inference rule we use greek letters to refer to arbitrary formulae that have (implicit) parentheses around them. To reiterate, every inference rule has the form

$$\frac{\phi_0, \phi_1, \ldots, \phi_m}{\psi_0, \psi_1 \ldots, \psi_n}$$

and is interpreted as saying that if all of the formulae $\phi_0, \ldots, \phi_m$ are true then you can conclude a conjunction of any number of the statements $\psi_0, \ldots, \psi_n$. In a MIZAR text, there is no need (and there is no way) to specify the name of a rule of inference. The MIZAR processor knows (almost all) the rules. Often, the writer of the proof will add comments indicating what inference rules were applied. An application of an inference rule in a MIZAR text is built according to the following general pattern.

$$\begin{array}{lll}
\texttt{k}_0\texttt{:} & \boxed{\quad\phi_0\quad} & \ldots \\
\texttt{k}_1\texttt{:} & \boxed{\quad\phi_1\quad} & \ldots \\
 & \ldots & \\
 & \ldots & \\
 & \ldots & \\
\texttt{k}_m\texttt{:} & \boxed{\quad\phi_m\quad} & \ldots
\end{array}$$

With all the formulae labelled $\texttt{k}_0$, $\texttt{k}_1, \ldots,$ $\texttt{k}_m$ available for reference we can add new lines to our argument, some of which look like:

$\boxed{\psi_0}$ `by` $\mathtt{k}_0$, $\mathtt{k}_1, \ldots,$ $\mathtt{k}_m$; `==` Name the rule here

or

$\boxed{\psi_1 \ \texttt{\&} \ \psi_3}$ `by` $\mathtt{k}_0$, $\mathtt{k}_1, \ldots,$ $\mathtt{k}_m$; `==` Name the rule here

or

$\boxed{\psi_0 \ \texttt{\&} \ \psi_1 \ \texttt{\&} \ \ldots \ \texttt{\&} \ \psi_n}$ `by` $\mathtt{k}_0$, $\mathtt{k}_1, \ldots,$ $\mathtt{k}_m$; `==` Name the rule here

Unfortunately, the notion of context somewhat complicates the use of the inference rules with regard to referencing premises. To understand context, we have to introduce the notion of visibility. But first let us look at some basic inference rules.

5.2 Basic Inference Rules

An argument is constructed one statement at a time, and has the shape of an optionally labeled formula followed by a justification. Since formulae vary widely in their form, we would expect to have a number of methods for adding new statements. For each of the logical connectives we have a rule that under the appropriate conditions lets us add a formula with that connective as the main one. Such rules are called *introduction* rules.

Conversely, since we may wish to make a reference to a formula of arbitrary shape in our argument, we have rules that under the appropriate conditions let us make a reference to the formula. These rules are called *elimination* rules.

We will start with a very special rule that is rarely useful.

5.2.1 Rewriting or Repetition, `RE`

Everyone would agree that once we establish a formula as true in a certain context, there is no need to repeat the argument in order to get the formula in another place, provided the context has not been changed.

$$\frac{\boxed{\texttt{0}}\boxed{}}{\boxed{\texttt{0}}\boxed{}}$$

Example:

```
     . . .
14: Knave[John] . . .
     . . .
17: Knave[John]  by 14;        == Rule: RE
     . . .
```

Label `14` must be visible when we make the reference (see 6.4). Also, we must be sure that the person named `John` at label `14` is the same person mentioned at label `17`.

5.2.2 Negation Elimination, `NE`

This rule is also known as double negation elimination.

$$\frac{\texttt{not not}\ \boxed{\texttt{0}}\boxed{}}{\boxed{\texttt{0}}\boxed{}}$$

In words: once we have the formula negated twice, we can drop both `not`s.

The box, $\boxed{0}\boxed{}$, may contain an arbitrary formula provided that

$$\texttt{not not}\ \boxed{0}\boxed{}$$

has the same structure as

$$\texttt{not not}\ (\ \boxed{0}\boxed{}\ )$$

To see this point note that the formula

```
not not Knave[John] or Knight[Mary]
```

cannot be used as a premise for applying `NE` as the main connective in the formula is `or` and not `not not`.

Example:

```
    . . .
14: not not Knave[John] . . .
    . . .
17: Knave[John]  by 14;        == Rule: NE
    . . .
```

5.2.3 Negation Introduction, `NI`

This rule is quite complex and will be defined later in section 6.2.

5.2.4 Conjunction Elimination, `CE`

$$\frac{\boxed{0}\boxed{\quad\quad} \;\&\; \boxed{1}\boxed{\quad\quad}}{\boxed{0}\boxed{\quad\quad},\; \boxed{1}\boxed{\quad\quad}}$$

The premise of this inference rule must have `&` as the main connective.

Example:

```
    . . .
4: Knave[John] & Knight[Mary] . . .
    . . .
5: Knight[Mary] by 4;        == Rule: CE
    . . .
7: Knave[John]  by 4;        == Rule: CE
    . . .
```

5.2.5 Conjunction Introduction, `CI`

$$\frac{\boxed{0}\boxed{\quad\quad},\; \boxed{1}\boxed{\quad\quad}}{\boxed{0}\boxed{\quad\quad} \;\&\; \boxed{1}\boxed{\quad\quad},\; \boxed{1}\boxed{\quad\quad} \;\&\; \boxed{0}\boxed{\quad\quad}}$$

When this inference rule is applied, the conclusion must have `&` as the main connective.

Example:

```
    . . .
2: Knave[John] . . .
    . . .
9: Knight[Mary] . . .
    . . .
24: Knave[John] & Knight[Mary] by 2, 9;        == Rule: CI
```

```
      . . .
    42: Knight[Mary] & Knave[John] by 2, 9;        == Rule: CI
      . . .
```

5.2.6 Disjunction Elimination, `DE`

Version 1:

$$\frac{\boxed{0}\boxed{}\ \texttt{or}\ \boxed{1}\boxed{},\ \texttt{not}\ \boxed{0}\boxed{}}{\boxed{1}\boxed{}}$$

Version 2:

$$\frac{\boxed{0}\boxed{}\ \texttt{or}\ \boxed{1}\boxed{},\ \texttt{not}\ \boxed{1}\boxed{}}{\boxed{0}\boxed{}}$$

These are two versions of the same rule. Although only one of them suffices, we introduce both, for convenience. This rule takes two premises, one of them must have `or` as the main connective, the other must be the negation of one of the disjuncts.

Example:

```
      . . .
    4: Knave[John] or not Knight[Mary] . . .
      . . .
    5: not not Knight[Mary] . . .
      . . .
    7: Knave[John]  by 4, 5;        == Rule: DE
      . . .
    9: not Knave[John] . . .
      . . .
    10: not Knight[Mary] by 4, 9;    == Rule: DE
      . . .
```

5.2.7 Disjunction Introduction, `DI`

$$\frac{\boxed{0}\boxed{}}{\boxed{0}\boxed{}\ \texttt{or}\ \boxed{1}\boxed{},\ \boxed{1}\boxed{}\ \texttt{or}\ \boxed{0}\boxed{}}$$

The conclusion of this rule must have `or` as the main connective.

Example:

```
      . . .
    5: not Knight[Mary] . . .
      . . .
    7: Knave[John] or not Knight[Mary] by 5;           == Rule: DI
      . . .
    10: not Knight[Mary] or Loves[John, Mary] by  5;    == Rule: DI
      . . .
```

5.2.8 Implication Elimination, `IE`, or Modus Ponens, `MP`

$$\frac{\boxed{\texttt{0}}\boxed{\quad\quad},\ \boxed{\texttt{0}}\boxed{\quad\quad}\ \texttt{implies}\ \boxed{\texttt{1}}\boxed{\quad\quad}}{\boxed{\texttt{1}}\boxed{\quad\quad}}$$

This is probably the best known rule of inference. It takes two premises, one must have `implies` as the main connective, and the other must be the antecedent of the implication. Then the conclusion is the consequent of the implication.

Example:

```
    . . .
5: Knight[John] . . .
    . . .
7: Knight[John] implies Loves[John, Mary] . . .
    . . .
10: Loves[John, Mary] by  5, 7;    == Rule: IE
    . . .
```

5.2.9 Implication Introduction, `II`

Unfortunately this rule is quite complex and will be defined later in section 6.1.

5.2.10 Equivalence Elimination, `EqE`

Version 1:

$$\frac{\boxed{\texttt{0}}\boxed{\quad\quad}\ \texttt{iff}\ \boxed{\texttt{1}}\boxed{\quad\quad},\ \boxed{\texttt{0}}\boxed{\quad\quad}}{\boxed{\texttt{1}}\boxed{\quad\quad}}$$

Version 2:

$$\frac{\boxed{\texttt{0}}\boxed{\quad\quad}\ \texttt{iff}\ \boxed{\texttt{1}}\boxed{\quad\quad},\ \boxed{\texttt{1}}\boxed{\quad\quad}}{\boxed{\texttt{0}}\boxed{\quad\quad}}$$

Although only one of this rules suffices, we introduce both for convenience. One of the premises must be an equivalence, that is, a formula with `iff` as the main connective.

Example:

```
    . . .
5: Knight[John] . . .
    . . .
7: Knight[John] iff Loves[John, Mary] & Knight[Mary] . . .
    . . .
10: Loves[John, Mary] & Knight[Mary] by  5, 7;    == Rule: EqE
    . . .
```

5.2.11 Equivalence Introduction, `EqI`

$$\frac{\boxed{0}\boxed{}\ \texttt{implies}\ \boxed{1}\boxed{},\ \boxed{1}\boxed{}\ \texttt{implies}\ \boxed{0}\boxed{}}{\boxed{0}\boxed{}\ \texttt{iff}\ \boxed{1}\boxed{}}$$

Example:

```
   . . .
3: Loves[John, Mary] & Knight[Mary] implies Knight[John] . . .
   . . .
7: Knight[John] implies Loves[John, Mary] & Knight[Mary] . . .
   . . .
8: Knight[John] iff Loves[John, Mary] & Knight[Mary] by 3, 7;
                                          == Rule: EqI
   . . .
```

5.2.12 Equality Elimination, `=E`

This rule will be defined later in section 7.3.

5.2.13 Equality Introduction, `=I`

$$\frac{}{\alpha\ \texttt{=}\ \alpha}$$

for arbitrary object name α. Note that this rule does not have any premise.

(ex07.mse) Example:

```
environ
        given x being aThing;
begin
        x = x;                 == Rule: =I
                               == Note: There is no ‘by’.
```

5.2.14 Rules for Quantifiers

The inference rules for quantifiers will be defined in section 7.

5.3 Annotated Proofs

(ex09.mse) Here is a simple proof in which we have annotated each line with the inference rule underlying its justification. This level of detail is unnecessary, and is here simply to illustrate how various rules can be used. In fact, as is illustrated by the last line, breaking up the reasoning into such simple pieces is unnecessary as Mizar understands how to reason about these kinds of formulas.

```
environ
    given x being Int;
    A1: A[x] or (B[x] implies C[x]);
    A2: not A[x] & (D[x] implies B[x]);
    A3: D[x];
begin
    1: not A[x] by A2;                   == Rule: CE
    2: D[x] implies B[x] by A2;          == Rule: CE
    3: B[x] by A3, 2;                    == Rule: MP
    4: B[x] implies C[x] by 1,A1;        == Rule: DE
    5: C[x] by 3,4;                      == Rule: MP

    6: B[x] & C[x] by 3, 5;              == Rule: CI
    7: B[x] or G[x] by 3;                == Rule: DI

    6: C[x] by A1, A2, A3; == directly using Mizar's proof capability
```

6 Hypothetical Reasoning

Hypothetical reasoning is a special form of proof in which we start by assuming the truth of a formula. The formula that we assume need not actually be true, thus the reasoning is termed hypothetical.

6.1 Implication Introduction, `II`

To justify a conditional of the form

$$\boxed{0}\boxed{}\ \texttt{implies}\ \boxed{1}\boxed{}$$

we must be able show that $\boxed{1}\boxed{}$ can be derived from $\boxed{0}\boxed{}$ (and possibly using also other formulae that are available for reference). The demonstration of this fact takes the form of a reasoning that starts with `now`. The basic form of this rule is:

$$\begin{array}{l}
\lambda\text{:}\ \ \texttt{now}\\
\quad\ \ \texttt{assume A: }\boxed{0}\boxed{};\\
\quad\ \ .\\
\quad\ \ .\\
\quad\ \ .\\
\quad\ \ \texttt{thus }\boxed{1}\boxed{}\quad \textit{Justification}\\
\texttt{end}\\
\hline
\boxed{0}\boxed{}\ \texttt{implies}\ \boxed{1}\boxed{}\ \texttt{by}\ \lambda
\end{array}$$

where, *Justification* is either simple, using `by`, or forms a proof (see below). The entire reasoning may have a label in front of `now` which is referenced when we apply the `II` rule. This rule is named *Implication Introduction*, as it lets us introduce a formula with `implies`
(ex10.mse) as the main connective. Consider the following example:

```
environ
 given x being PERSON;
 Ax1: P[x] implies Q[x];
 Ax2: Q[x] implies R[x];
begin

 1: now
      assume A: P[x];
            a1: Q[x] by A, Ax1;           == Rule: IE
      thus R[x] by a1, Ax2;               == Rule: IE
    end;
 P[x] implies R[x] by 1;                  == Rule: II

 2: now
        assume A: (P[x] implies Q[x]) & (Q[x] implies R[x]) & P[x];
              a1: P[x] by A;              == CE
              a2: P[x] implies Q[x] by A; == CE
              a3: Q[x] by a1, a2;         == IE
            a4: Q[x] implies R[x] by A; == CE
        thus R[x] by a3, a4;            == IE
    end;

  ( (P[x] implies Q[x]) & (Q[x] implies R[x]) & P[x] ) implies R[x]
                                        by 2;     == II
```

6.2 Proof by Contradiction

Remember that we stated that only statements that are true in their contexts can appear in an argument. But what happens when both a statement and its negation are true in the same context? This is a contradiction, and indicates an internal inconsistency in the current context. We have a special formula, `contradiction`, that can be derived using the following inference rule:

6.2.1 Contradiction Introduction, `ContrI`

$$\frac{\boxed{0}\boxed{},\ \texttt{not}\ \boxed{0}\boxed{}}{\texttt{contradiction}}$$

The always false, constant formula `contradiction` and the `ContrI` rule of inference are introduced for convenience, we could work without them.

Example:

```
        . . .
    14: Knight[John] . . .
        . . .
    25: not Knight[John] . . .
        . . .
        contradiction by 14, 25;       == ContrI
        . . .
```

6.2.2 Negation Introduction, `NI`

It should be impossible to establish `contradiction` (in any consistent proof system) by correct reasoning from consistent premises. If you do, you are permitted to conclude that the assumptions you made which led to `contradiction` must be false.

This form of argument was illustrated by our knights and knaves example. In this kind of argument we typically begin by assuming that some statement $\boxed{0}\boxed{}$ is true, and then proceed to reason on the basis of this assumption. If we encounter some kind of contradiction or absurd situation we stop reasoning and conclude that, since $\boxed{0}\boxed{}$ leads to nonsense, $\boxed{0}\boxed{}$ cannot be true and so **`not`** $\boxed{0}\boxed{}$ must be true. This kind of reasoning is called *proof by contradiction* and forms the essence of the *Negation Introduction* rule.

$$\begin{array}{l}
\lambda\text{:}\quad \texttt{now} \\
\qquad \texttt{assume A: } \boxed{0}\boxed{}; \\
\qquad . \\
\qquad . \\
\qquad . \\
\qquad \texttt{thus contradiction} \quad \textit{Justification} \\
\texttt{end} \\
\hline
\qquad\qquad \texttt{not } \boxed{0}\boxed{} \texttt{ by } \lambda
\end{array}$$

Now you see why this form of proof is named *Negation Introduction*; it provides us with the
(ex12.mse) means to obtain a negated formula (if possible). For example:

```
environ
    given glass being CONTAINER;
    given juice being DRINK;
    A1: Full[glass] implies Fresh[juice];
    A2: not Fresh[juice];

begin       == We show that: not Full[glass]

   S: now
        assume A: Full[glass];
               1: Fresh[juice] by A, A1;  == IE
        thus contradiction by 1,A2;       == ContrI
    end;
    not Full[glass] by S;                 == NI
```

The next example illustrates how to employ a proof by contradiction when proving a formula
(ex14.mse) that does not have `not` as its main connective.

```
environ
    given glass being CONTAINER;
begin  == We will demonstrate that the following weird formula is always true:
       ==         (not Full[glass] implies Full[glass]) implies Full[glass]

    1: now
        assume a: not Full[glass] implies Full[glass];
               S: now    == We prepare for Negation Introduction
                    assume b: not Full[glass];
                           c: Full[glass] by a, b;            == MP
                    thus contradiction by c, b;               == ContrI
                  end;

               d: not not Full[glass] by S;                   == NI
        thus Full[glass] by d                                 == NE
    end;

    (not Full[glass] implies Full[glass]) implies Full[glass] by 1; == II
```

In the inner `now` we have showed that `not Full[glass]` leads to `contradiction`. From the reasoning we obtained `not not Full[glass]` using the `NI` rule. Then we stripped off the double negation using the `NE` rule. To simplify matters, we introduce an alternative form of proof by contradiction. This alternative form of the `NI` rule will be named proof by contradiction with the abbreviation `PbyC`.

```
      λ:  now
           assume Contra:  not |0|    |;
           .
           .
           .
           thus contradiction   Justification
          end
  ---------------------------------------------
                  |0|    | by λ
```

Do you see that this rule is superfluous? The same effect always can be achieved by proving `not not` $\boxed{0}\boxed{}$ using `NI` and then stripping off the double negation by `NE`.

We will conclude this section with an excursion into a non-existing world. It is the world where `contradiction` has been assumed as one of the axioms. This would make `contradiction` true. But since `contradiction` is false, it cannot be so; such a world does not exist, and we can expect a lot of surprising results. (Note that despite the fact that the world does not exist—false assumptions—we can talk about it. This should not be surprising at all, the politicians do it all the time, and we do it only in this place.)

(ex13.mse) We will prove that in such a world every formula can be proven.

```
environ
        StrangeWorld: contradiction;
begin
        1: now
                assume not ArbitraryFormula[];
                thus contradiction by StrangeWorld     == RE
        end;

        ArbitraryFormula[] by 1;                       == PbyC
```

So, if you manage to get a contradiction in any reasoning then you can prove whatever you wish. This is the reason we try to avoid contradictory worlds at all cost. In using the `NI` rule, the obtained `contradiction` is usually blamed on the locally made assumption.

6.3 Derived Rules Of Inference

The basic rules of inference discussed in section 5.2 and the hypothetical reasoning rules (`II` and `NI`) are sufficient for conducting arguments. However, using only this small set of rules makes arguments quite lengthy. Therefore, we introduce the notion of a *derived rule of inference* which we will use instead of the lengthy sequence of basic rules. Look at an example. (ex16.mse)

```
environ
    given x, cow being THING;
    1: Wood[x] implies CanCook[x, cow];
    2: CanCook[x, cow] implies Eat[cow];
    3: Wood[x];
begin
        step1: CanCook[x, cow] by 1, 3;          == IE
        step2: Eat[cow] by 2, step1;             == IE
```

In any similar situation and using only the basic rules we will have to repeat the two steps of the Modus Ponens application. However, we will say that by virtue of the above example we are free to use the following derived rule of inference:

6.3.1 Modus Ponens Twice

$$\frac{\boxed{0}\boxed{\quad\quad} \texttt{ implies } \boxed{1}\boxed{\quad\quad},\ \boxed{1}\boxed{\quad\quad} \texttt{ implies } \boxed{2}\boxed{\quad\quad},\ \boxed{0}\boxed{\quad\quad}}{\boxed{2}\boxed{\quad\quad}}$$

One can certainly think about a number of similar rules, one for each number of repeated Modus Ponens. Now, do we have to remember all these derived rules? The answer is *no*. The MIZAR processor does not require us to specify any rule of inference in our arguments. It is the processor that checks whether any step we made in our reasoning can be obtained by a sequence of applications of basic inference steps.

Below, we list a number of the more popular derived inference rules. We will require you to annotate some of your derivations. Understanding the rules of inference is crucial to learning proof methods.

6.3.2 Double Negation Introduction

$$\frac{\boxed{0}\boxed{\quad\quad}}{\texttt{not not } \boxed{0}\boxed{\quad\quad}}$$

6.3.3 Reverse Implication Elimination or Modus Tollens (MT)

$$\frac{\boxed{0}\boxed{\quad\quad} \texttt{ implies } \boxed{1}\boxed{\quad\quad},\ \texttt{not } \boxed{1}\boxed{\quad\quad}}{\texttt{not } \boxed{0}\boxed{\quad\quad}}$$

The above rule corresponds to the following argument in which we use only basic infer- (ex17.mse) ence rules.

```
environ
    given 172 being COURSE;
    1: Goodmarks[172] implies Pass[172];
    2: not Pass[172];
begin

  1: now
       assume  step1: Goodmarks[172];
               step2: Pass[172] by step1, 1;            == IE
       thus    step3: contradiction by 2, step2;        == ContrI
  end;

not Goodmarks[172]  by 1;                               == PbyC
```

6.3.4 Excluded Middle

$$\frac{}{\boxed{\texttt{0}}\boxed{\quad\quad}\ \texttt{or not}\ \boxed{\texttt{0}}\boxed{\quad\quad}}$$

The above rule corresponds to the following argument in which we use only basic rules (ex18.mse) of inference:

```
environ
        given s being STATEMENT;
        == No other premises.
begin

  1: now
       assume step1: not (True[s] or not True[s]);

         step2: now
                  assume step3: not True[s];
                         step4: True[s] or not True[s] by step3;  == DI
                  thus   step5: contradiction by step1, step4;    == ContrI
                end;

         step6: True[s] by step2;                                 == PbyC
         step7: True[s] or not True[s] by step6;                  == DI
       thus contradiction by step1, step7                         == ContrI
  end;

  True[s] or not True[s] by 1                                     == PbyC
```

6.3.5 Chained Implication

$$\frac{\boxed{0}\boxed{}\ \texttt{implies}\ \boxed{1}\boxed{},\ \boxed{1}\boxed{}\ \texttt{implies}\ \boxed{2}\boxed{}}{\boxed{0}\boxed{}\ \texttt{implies}\ \boxed{2}\boxed{}}$$

The same effect as from applying the Chained Implication rule can be obtained using only (ex19.mse) basic rules:

```
environ
        given x, cow being THING;
        1: Wood[x] implies CanCook[x, cow];
        2: CanCook[x, cow] implies Eat[cow];
begin
  1: now
       assume  step1: Wood[x];
               step2: CanCook[x, cow] by 1, step1;   == MP
       thus    step3: Eat[cow] by 2, step2;          == MP
  end;

Wood[x] implies Eat[cow] by 1                        == II
```

You are strongly encouraged to demonstrate that the following derived rules of inference correspond to a sequence of basic inferences. (We do not introduce standard acronyms for the names of derived rules.)

6.3.6 Reverse Implication

$$\frac{\boxed{0}\boxed{}\ \texttt{implies}\ \boxed{1}\boxed{}}{\texttt{not}\ \boxed{1}\boxed{}\ \texttt{implies not}\ \boxed{0}\boxed{}}$$

$$\frac{\texttt{not}\ \boxed{1}\boxed{}\ \texttt{implies not}\ \boxed{0}\boxed{}}{\boxed{0}\boxed{}\ \texttt{implies}\ \boxed{1}\boxed{}}$$

6.3.7 Case Analysis

$$\frac{\boxed{0}\boxed{}\ \texttt{implies}\ \boxed{2}\boxed{},\ \boxed{1}\boxed{}\ \texttt{implies}\ \boxed{2}\boxed{}}{\boxed{0}\boxed{}\ \texttt{or}\ \boxed{1}\boxed{}\ \texttt{implies}\ \boxed{2}\boxed{}}$$

$$\frac{\boxed{0}\boxed{}\ \texttt{or}\ \boxed{1}\boxed{},\ \boxed{0}\boxed{}\ \texttt{implies}\ \boxed{2}\boxed{},\ \boxed{1}\boxed{}\ \texttt{implies}\ \boxed{2}\boxed{}}{\boxed{2}\boxed{}}$$

6.3.8 Equivalence Elimination

$$\frac{\boxed{0}\boxed{}\ \texttt{iff}\ \boxed{1}\boxed{}}{\boxed{0}\boxed{}\ \texttt{implies}\ \boxed{1}\boxed{},\ \boxed{1}\boxed{}\ \texttt{implies}\ \boxed{0}\boxed{}}$$

$$\dfrac{\boxed{0\mid\quad}\ \texttt{iff}\ \boxed{1\mid\quad}}{\texttt{(}\boxed{0\mid\quad}\ \texttt{\&}\ \boxed{1\mid\quad}\texttt{) or (not}\ \boxed{0\mid\quad}\ \texttt{\& not}\ \boxed{1\mid\quad}\texttt{)}}$$

6.3.9 Equivalence Introduction

$$\dfrac{\texttt{(}\boxed{0\mid\quad}\ \texttt{\&}\ \boxed{1\mid\quad}\texttt{) or (not}\ \boxed{0\mid\quad}\ \texttt{\& not}\ \boxed{1\mid\quad}\texttt{)}}{\boxed{0\mid\quad}\ \texttt{iff}\ \boxed{1\mid\quad}}$$

6.3.10 Divergence

$$\dfrac{\boxed{0\mid\quad}\ \texttt{iff}\ \boxed{1\mid\quad}\texttt{, not}\ \boxed{0\mid\quad}}{\texttt{not}\ \boxed{1\mid\quad}}$$

6.3.11 De Morgan's Laws

See 7.9 for the analogous rules involving quantifiers.

$$\dfrac{\boxed{0\mid\quad}\ \texttt{\&}\ \boxed{1\mid\quad}}{\texttt{not (not}\ \boxed{0\mid\quad}\ \texttt{or not}\ \boxed{1\mid\quad}\texttt{)}}$$

$$\dfrac{\boxed{0\mid\quad}\ \texttt{or}\ \boxed{1\mid\quad}}{\texttt{not (not}\ \boxed{0\mid\quad}\ \texttt{\& not}\ \boxed{1\mid\quad}\texttt{)}}$$

$$\dfrac{\texttt{not (}\boxed{0\mid\quad}\ \texttt{\&}\ \boxed{1\mid\quad}\texttt{)}}{\texttt{not}\ \boxed{0\mid\quad}\ \texttt{or not}\ \boxed{1\mid\quad}}$$

$$\dfrac{\texttt{not(}\boxed{0\mid\quad}\ \texttt{or}\ \boxed{1\mid\quad}\texttt{)}}{\texttt{not}\ \boxed{0\mid\quad}\ \texttt{\& not}\ \boxed{1\mid\quad}}$$

6.3.12 More Disjunction Elimination

$$\dfrac{\texttt{not}\ \boxed{0\mid\quad}\ \texttt{or}\ \boxed{1\mid\quad}\texttt{,}\ \boxed{0\mid\quad}}{\boxed{1\mid\quad}}$$

$$\dfrac{\boxed{0\mid\quad}\ \texttt{or not}\ \boxed{1\mid\quad}\texttt{,}\ \boxed{1\mid\quad}}{\boxed{0\mid\quad}}$$

6.3.13 Another Implication Elimination

$$\frac{\boxed{0}\boxed{\qquad}\ \texttt{implies}\ \boxed{1}\boxed{\qquad}}{\texttt{not}\ \boxed{0}\boxed{\qquad}\ \texttt{or}\ \boxed{1}\boxed{\qquad}}$$

6.3.14 More Implication Introduction

$$\frac{\texttt{not}\ \boxed{0}\boxed{\qquad}\ \texttt{or}\ \boxed{1}\boxed{\qquad}}{\boxed{0}\boxed{\qquad}\ \texttt{implies}\ \boxed{1}\boxed{\qquad}}$$

$$\frac{\boxed{1}\boxed{\qquad}}{\boxed{0}\boxed{\qquad}\ \texttt{implies}\ \boxed{1}\boxed{\qquad}}$$

$$\frac{\texttt{not}\ \boxed{0}\boxed{\qquad}}{\boxed{0}\boxed{\qquad}\ \texttt{implies}\ \boxed{1}\boxed{\qquad}}$$

6.3.15 Referencing Distributed Statements

The construction *DistributedStatement* with syntax

[*LabelId* :] `now` *Reasoning* `end`

provides us with an alternative way of recording formulas together with their justification. Distributed—because pieces of a formula are distributed along the reasoning. We will discuss the details of this distribution in section 8.4. Here, we would like to point out that distributed formulas can be referenced in a way similar to the regular ones. Consider a reasoning:

```
      . . .
1': now
       . . .
       . . .
       . . .
     end;
```

We can always write a compact sentence

```
      . . .
1'': . . . by 1';
      . . .
```

which will be accepted as correct and in the text that follows, it does not matter whether we make a reference to `1''` or to `1'`—both these labels refer to the same formula. For reference, a formula need not be recorded in a compact way; the distributed version of the formula suffices.

We can say that the `II` rule provides us with the distributed means of recording a conditional: the connective `implies` need not be mentioned at all. Analogous remarks apply for the `NI` rule.

6.4 The Notion Of Visibility in a Reasoning

A Mizar text usually contains many statements and reasonings. At every step in a reasoning it is important to know what formulas and objects can or cannot be used to establish the next step. To this end we introduce the notion of *visibility* of identifiers, for both object identifiers and formula labels.

Recall that a Mizar *Article* consists of two parts: the *Environment* and the *TextProper*. The environment contains two main kinds of statements: axioms, which are formulas with a label, and global constant declarations, which specify objects of a specific sort whose meaning is fixed throughout the proof. Global constant declarations are signaled by the keyword `given`, while axioms are simply stated without justification. Every free object identifier in an axiom must be a previously declared global constant.

In the text proper we also encounter two main kinds of statements: formulas together with their justifications and introductions of names for objects. (Introductions of object identifiers are signified by the keywords `let` and `consider`, to be introduced later.) Labels name formulas. Any identifier can be introduced more than once as a name of an object. As our creative capabilities when inventing identifiers are usually limited, we tend to repeat identifiers for different purposes in different places of the text. In Mizar, unlike in many programming languages, the same identifier can be freely reused to denote two different things even in the same place. The following text is correct. (ex23.mse)

```
environ
        given A being A;
  A: A[A] implies not A[A];

begin
        not A[A] by A        == Mizar is skillful enough to see it.
```

The principle is simple: identifiers naming different kinds of objects are never in conflict. Therefore, in the same place you can use the same identifier, in our case `A`, to name: a constant (or a bound variable, but not both), a sort, a formula, and a predicate.

The text encompassed by `now` and `end` forms a new scope of identifiers. Any identifier (say a label) introduced in the reasoning overrides the same identifier introduced outside the reasoning and renders the original identifier inaccessible in the reasoning. (We hope that you are comfortable with scoping rules for some programming language.)

Suppose that you are at some line λ in a text. The general method for determining what formulas and object identifiers are visible at line λ is as follows:

1. The only visible formulas and object identifiers at line λ are those associated with syntactically complete statements that appear in their entirety before line λ.

2. Of the formulas and object identifiers allowed by 1 above, only the actual formulas being justified, or the object identifiers being introduced are visible at line λ. Nothing appearing in the justification of a formula is visible outside the justification.

3. Of the formulas and object identifiers allowed by 2 above, only the most recently named formula is visible among the formulas with the same label; and only the most recently introduced occurrence of an object identifier is visible.

4. Of the formulas allowed by 3 above, if line λ introduces an object identifier α, then only those formulas that *do not* have α free in them are properly visible. We put the qualification—properly—as the formulas may be visible with the confusion between the two αs completely unclear to the human reader (and the writer as well).

(ex25.mse) The following example illustrates visibility with respect to formulas:

```
environ
    A1: P[] implies Q[];
    A2: P[] or not R[];
    A3: not R[] implies S[];
begin
                == At this point A1, A2, A3 are all visible
    1: now
                == At this point A1, A2, A3 are all visible, but 1 is not,
                == as the statement it labels is not completed yet.
                == 2, 3 are not visible because they follow this point
          assume 2: not P[];
                 3: not R[] by A2, 2;
                == At this point, 2 and 3 are now visible.
          thus S[] by 3,A3;
       end;

    not P[] implies S[] by 1;
       == At this point A1, A2, A3, and 1 are all visible.
       == 2, 3 are not because they are inside a justification.

    4: now
          assume 5: R[];
                 6: now
                       assume 7: P[];
                       thus R[] by 5;
                    end;
                 8: now
                       == A1, A2, A3, 1, 5, and 6 are visible.
                       == 7 is not because it is inside a justification.
                       == 4 is not because we are inside the ‘formula‘ it
                       == labels.
                     assume 9: not P[];
                     thus R[] by 5;
                    end;
          thus (P[] implies R[]) & (not P[] implies R[]) by 6,8;
       end;

    R[] implies (P[] implies R[]) & (not P[] implies R[]) by 4;

       == This next statement redefines A1, so the previous
       == formula referred to by A1 is no longer visible.
    A1: not R[] or R[];
```

6.5 Examples

(ex20.mse) Let us prove that: `(P[] implies Q[]) & (not P[] implies Q[]) implies Q[]`

```
environ   == Note the empty environment.
begin
  1: now
       assume A: (P[] implies Q[]) & (not P[] implies Q[]);
              1: P[] or not P[];                          == Excluded Middle
              2: P[] implies Q[] by A;                    == CE
              3: not P[] implies Q[] by A;                == CE
              4: (P[] or not P[]) implies Q[] by 1, 2, 3;   == Case Analysis
       thus Q[] by 1, 4;                                  == MP
     end;

( (P[] implies Q[]) & (not P[] implies Q[]) ) implies Q[] by 1     == II
```

Here is an example that proves that

$$\boxed{0}\boxed{}\ \texttt{implies}\ \boxed{1}\boxed{}$$

always has the same logical value as

$$\texttt{not}\ \boxed{0}\boxed{}\ \texttt{or}\ \boxed{1}\boxed{}$$

(ex15.mse) Thus, we prove that the two formulas are equivalent.

```
environ
    given today being DAY;
begin

  one_way: now
     assume A: Raining[today] implies Cloudy[today];

            1: Raining[today] or not Raining[today];     == Excluded Middle

            2: now
                   assume A2: not Raining[today];
                   thus not Raining[today] or Cloudy[today] by A2;    == DI
               end;

            3: now
                   assume A3: Raining[today];
                     3a: Cloudy[today] by A, A3;                      == MP
                   thus not Raining[today] or Cloudy[today] by 3a;    == DI
                end;

     thus not Raining[today] or Cloudy[today] by 1,2,3; == Case Analysis
  end;

  other_way: now
     assume B: not Raining[today] or Cloudy[today];
```

```
            1’: now
                  assume B1: Raining[today];
                  thus Cloudy[today] by B, B1;                   == More DE
                end;
      thus Raining[today] implies Cloudy[today] by 1’                 == II
   end;

 (Raining[today] implies Cloudy[today])
                                    iff
                                        not Raining[today] or Cloudy[today]
                                             by one_way, other_way   == EqI
```

Note how we have made references to distributed formulas above. We save space by not recording the compact versions of the formulas that will be obtained from the reasonings labelled one_way and other_way applying the II rule. The compact formula is needed in a conclusion, thus we have to apply the the II rule at one point.

(ex21.mse) Here is another example:

```
environ
    given S being STUDENT;
    given M being Money;
begin

  1’: now
       assume A:( Need[M] implies Work[S] & Save[M] );

             1: now
                 assume A1: Need[M];
                        1a: Work[S] & Save[M] by A, A1;                == IE
                 thus Work[S] by 1a;                                   == CE
                end; == Do you see what we have demonstrated?

             2: now
                 assume A1: not Save[M];
                        2a: now
                             assume A2: Need[M];
                                    2b: Work[S] & Save[M] by A, A2;    == IE
                             thus Save[M] by 2b;                       == CE
                            end; == Do you see what we have demonstrated?
                 thus not Need[M] by A1, 2a;                           == MT
                end; == Do you see what we have demonstrated?

            1’: Need[M] implies Work[S] by 1;                          == II
            2’: not Save[M] implies not Need[M] by 2;                  == II

       thus (Need[M] implies Work[S]) & (not Save[M] implies not Need[M])
                                         by 1’, 2’;                    == CI
      end;

 (Need[M] implies Work[S] & Save[M])
                               implies
```

```
                  (Need[M] implies Work[S]) & (not Save[M] implies not Need[M])
                                                        by 1'           == II
```

Of course Mizar's knowledge of propositional logic makes all of these example steps unnecessary, and that is why step `2a` in the proof above could be abbreviated. We go into such detail with these simple proofs only to illustrate how the inference rules work, not to indicate that this is how you should do a complex proof—it is simply that the details of a complex proof get in the way of understanding the basic inferences.

7 Inference Rules for Quantified Formulas and Equality

7.1 Bound Variable Renaming

We have already mentioned that the names of bound variables do not matter. Everybody agrees that

```
for x being T holds P[x,y]
```

is equivalent to

```
for z being T holds P[z,y]
```

Sometimes we would like to rename some of the identifiers in a formula in order to make it easier to read. How do we ensure that our renaming does not change the meaning of the original formula?

Given a formula ϕ, we may change the name of a bound variable in ϕ and obtain a formula ϕ' equivalent to ϕ provided no occurrence of an object identifier that was free in ϕ becomes bound in ϕ', and every occurrence of a bound object identifier in ϕ has the same binding in ϕ'. (Note, it is not enough that bound object identifiers remain bound, they must remain bound to the same quantifier.)

Whenever we rename bound variables we refer to the rule of inference called *Bound Variable Renaming.* Bound variable renaming does not require any justification in MIZAR texts.

Let us look at some examples.

1. `for z being T holds P[z,y]`

   is equivalent to

   ```
   for aaaa being T holds P[aaaa,y]
   ```

   and is *not* equivalent to

   ```
   for y being T holds P[y,y]
   ```

   as a formerly free object identifier (the `y` in `P[z,y]`) has become bound.

2. `ex z being T st for x being T holds P[x,y,z]`

   is *not* equivalent to

   ```
   ex z being T st for z being T holds P[z,y,z]
   ```

   as a free object identifier `x` in `for x being T holds P[x,y,z]` has become bound in `for z being T holds P[z,y,z]`.

3. `ex z being T st for x being T holds P[x,y,z]`

   is *not* equivalent to

   ```
   ex y being T st for x being T holds P[x,y,y]
   ```

   as a free object identifier `y` has become bound.

7.2 Proper Free Substitution

Inference rules for quantifiers involve renaming of object identifiers and a textual operation on formulas, called *proper free substitution.* This is its definition:

> We say the formula ψ comes from formula ϕ by *proper free substitution of object identifier β for object identifier α* if
>
> 1. ψ is exactly like ϕ except that wherever ϕ has **free occurrences** of α, ψ has **free occurrences** of β, and
> 2. the sort of β is the same as the sort of α.

It is important to remember that proper free substitution is *not* an inference rule. It is a rule for editing formulas textually.

For example, the formula

```
Knave[Fred] & Knight[Wilma] implies (ex Fred being PERSON st Tells[Fred,s])
```

becomes, under proper free substitution of `Wilma` for `Fred`, the formula

```
Knave[Wilma] & Knight[Wilma] implies (ex Fred being PERSON st
                         Tells[Fred,s])
```

This in turn becomes, under proper free substitution of `Hello` for `s`, the formula

```
Knave[Wilma] & Knight[Wilma] implies (ex Fred being PERSON st
                      Tells[Fred,Hello])
```

We use the notation $\boxed{}_{\alpha\leftarrow\beta}$ to stand for the formula resulting from the proper free substitution of β for α in formula $\boxed{}$. Another way to think of this notation is of replacing every free occurrence of α *by* β without introducing any new bound occurrences of β. Note that there is no notation for proper free substitution that is consistently used by all authors of logic texts, so it is always important to determine the convention used for each particular work.

This notation lets us define three more inference rules.

7.3 Equality Elimination, =E

Equality elimination is the rule that we apply when making a reference to a known equality between two objects.

$$\frac{\alpha\ \texttt{=}\ \beta\texttt{,}\ \boxed{0}\boxed{}}{\boxed{1}\boxed{}} \quad \text{provided } \boxed{0}\boxed{}_{\alpha\leftarrow\beta} \text{ is identical to } \boxed{1}\boxed{}_{\alpha\leftarrow\beta}$$

This requires some explanation. The intended meaning of this rule is that we can exchange some or all free occurrences of α and β in a formula provided $\alpha = \beta$. The condition attached to the formula says just that. It might be easier to understand the formula if you look at it this way. Knowing that $\alpha = \beta$ we may think that α is just another name of the same object named by β. So, let us choose β as the only name that we use, α is just an alias.

Examples.

(ex26.mse) 1. Equality is symmetric.

```
environ
           given a, b being THING;
        1: a = b;
begin
           b = a by 1;  == =E
```

This example may look cryptic. The two premises needed by the inference rule are formed by the same formula `a=b`. And indeed, $\boxed{0}\boxed{\texttt{a=b}}_{\texttt{a}\leftarrow\texttt{b}}$ is identical to $\boxed{1}\boxed{\texttt{b=a}}_{\texttt{a}\leftarrow\texttt{b}}$ as both of them are `b=b`.

(ex27.mse) 2. Equality is transitive.

```
environ
           given a, b, c being THING;
        1: a = b;
        2: b = c;
begin
           a = c by 1, 2;  == =E
```

Here we see that $\boxed{0}\boxed{\texttt{b=c}}_{\texttt{a}\leftarrow\texttt{b}}$ is identical to $\boxed{1}\boxed{\texttt{a=c}}_{\texttt{a}\leftarrow\texttt{b}}$.

(ex28.mse) 3.

```
environ
           given a, b being THING;
        1: P[a];
        2: not P[b];
begin
  R: now
       assume step1: a = b;
```

```
                step2: not P[a] by step1, 2;          == =E
          thus  step3: contradiction by 1, step2;     == ContrI
    end;

    a <> b by R;                                      == PbyC
```

4. MIZAR automatically processes equality as a reflexive, symmetric and transitive relation. (Reflexivity is expressed by the Equality Introduction (`=I`) rule introduced earlier.) Therefore, the following argument requires only one step of inference. (ex29.mse)

```
environ
           given a, b, c being THING;
        1: P[a];
        2: not P[c];
        3: a = b;
begin
           b <> c by 1, 2, 3;     == No single inference rule suffices.

           == Mizar can figure out which basic rules are involved.
```

It is a worthwhile exercise to write the above using only single inference rules.

7.4 Universal Elimination, UE

Universal elimination is used to convert a general statement about any object into a specific statement about a particular object.

$$\frac{\texttt{for } \alpha \texttt{ being } \tau \texttt{ holds } \boxed{\;\;\;\;\;\;}}{\boxed{\;\;\;\;\;\;}_{\alpha\leftarrow\beta}}$$

where β is any visible object identifier of sort τ.

This rule is sometimes called universal instantiation, or universal specialization, because it specializes a general statement about any object into a specific statement about a particular object. It represents our intuition that what holds for everything must hold for any
(ex30.mse) specific thing. Look at these inferences.

```
environ
    given Fred being PERSON;
    given Hello being STATEMENT;

    KnightSpeak: for y being PERSON holds
                    for s being STATEMENT holds
                       Knight[y] & Tells[y,s] implies True[s];

begin

    == Specialize the y to Fred.

    1: for s being STATEMENT holds
          Knight[Fred] & Tells[Fred,s] implies True[s]
                                            by KnightSpeak;     == UE

    == Further Specialize the s in 1 to Hello.

    2: Knight[Fred] & Tells[Fred,Hello] implies True[Hello] by 1; == UE

    == You must instantiate in the order that the quantifiers occur
    == thus attempting to specify s to Hello will not work.

    3: for y being PERSON holds
        Knight[y] & Tells[y,Hello] implies True[Hello] by KnightSpeak;

    == That is, when you have a list of universal quantifiers, only the
    == leading ones can be eliminated.
    == In order to achieve the inner universal quantifier elimination you
    == have to use Universal Introduction, see below.
```

7.5 Existential Introduction, `ExI`

This next rule corresponds to our intuition that if a formula holds for a particular thing, then we can discard some information and simply say that the formula holds for some thing. Using this rule we obtain a weaker formula than the premise and in the process we lose information.

Suppose that you have the formula `A[x] & B[x]` in some context, where `x` is of sort `THING`. This means that there is a precise object, named `x`, which satisfies the formula. We can then certainly write:

```
ex y being THING st A[y] & B[y]
```

This formula is read as "There exists some object `y` satisfying `A[y] & B[y]`". We know that this is true at least in the case that `y` denotes the same object as `x`. There may also be objects other than the one denoted by `x` which also satisfy the formula.

But we also know that the following formula is true for exactly the same reasons:

```
ex y being THING st A[y] & B[x]
```

Note how only the argument to predicate `A` was changed.

In general we can do the following kind of inference.

$$\frac{\boxed{}_{\alpha\leftarrow\beta}}{\texttt{ex } \alpha \texttt{ being } \tau \texttt{ st } \boxed{}}$$

where β is a visible object identifier of sort τ.

This rule is sometimes called *Existential Generalization.* Note the particular form of this rule.

- The premise must not have free αs. Note that the formula of the form $\boxed{}_{\alpha\leftarrow\beta}$ does not have free αs.
- By applying the rule to the premise you obtain a new formula in which you may rename **zero or more occurrences** of object identifier β to α and then existentially quantify the αs. (Note not necessarily all!).

For example, here are different existential statements that can be concluded from the same formula. (ex31.mse)

```
environ
    given x,y being THING;
    A: P[x,y,x];
begin

                == Generalize with respect to y.
    ex a being THING st P[x,a,x] by A;                      == ExI

                == Generalize with respect to x.
    ex b being THING st P[b,y,b] by A;                      == ExI
```

```
                == Generalize with respect to the first x.
    ex c being THING st P[c,y,x] by A;                        == ExI

                == Generalize with respect to nothing.
    ex d being THING st P[x,y,x] by A;                        == ExI
```

Here is an example of using universal elimination and existential introduction in an argument. (ex32.mse)

```
environ
    given Fred being PERSON;
    given Hello being STATEMENT;

    A1: for x being PERSON holds
           ex s being STATEMENT st
             (Tells[x,s] & Tells[x,Hello] & s <> Hello & True[s]);
begin
          == Here we universally instantiate, substituting Fred for x
          == and eliminating the universal quantifier.
1: ex s being STATEMENT st
     (Tells[Fred,s] & Tells[Fred,Hello] & s <> Hello & True[s]) by A1; == UE

          == Existential generalization, replace all occurrences
          == of Hello by t, and add the existential quantifier
2: ex t being STATEMENT st
     ex s being STATEMENT st
       (Tells[Fred,s] & Tells[Fred,t] & s <> t & True[s]) by 1;        == ExI

          == Existentially generalize, replace only the first
          == of Fred occurrences by y, and add the existential quantifier.
3: ex y being PERSON st
     ex t being STATEMENT st
       ex s being STATEMENT st
         (Tells[y,s] & Tells[Fred,t] & s <> t & True[s]) by 2;         == ExI
```

7.6 Examples of Universal Elimination, and Existential Introduction

(ueandei.mse)

```
environ
    given Fred being PERSON;
    given Hello being STATEMENT;

    KnightSpeak: for y being PERSON for s being STATEMENT
                   holds Knight[y] & Tells[y,s] implies True[s];

    A1: Knight[Fred];
    A2: Tells[Fred, Hello];

begin
        == Here is an example of Universal Instantiation.
```

```
1: for s being STATEMENT holds
    Knight[Fred] & Tells[Fred,s] implies True[s] by KnightSpeak;

    == And here is an example of Existential Generalization.
1a: ex s being STATEMENT st Tells[Fred,s] by A2;
1b: ex x being PERSON, s being STATEMENT st Tells[x,s] by A2;
1c: ex x being PERSON st Tells[x,Hello] by A2;

    == This is also correct.
1d: ex x being KNIGHT, s being STATEMENT st Tells[Fred,Hello] by A2;

KnightSpeak1: for t being STATEMENT for x being PERSON
                holds Knight[x] & Tells[x,t] implies True[t] proof
        let t be STATEMENT;
            == now we have to show the "for x ..." part
        let x be PERSON;
        thus Knight[x] & Tells[x,t] implies True[t] by KnightSpeak;
end;

2: for y being PERSON holds
    Knight[y] & Tells[y,Hello] implies True[Hello] by KnightSpeak1;

2a: for y being PERSON holds
    Knight[y] & Tells[y,Hello] implies True[Hello] proof
        let y be PERSON;
        1: for s being STATEMENT holds
             Knight[y] & Tells[y,s] implies True[s] by KnightSpeak;
        thus
            Knight[y] & Tells[y,Hello] implies True[Hello] by 1;
end;

3: Knight[Fred] & Tells[Fred,Hello] implies True[Hello] by KnightSpeak;

4: True[Hello] by A1, A2, 3;
```

7.7 Universal Introduction, `UI`

To introduce a universal quantifier we do another kind of hypothetical reasoning, called *universal generalization* or *universal introduction*. The intuition here is that to prove that some formula holds for all objects we show that it holds for any arbitrarily selected but fixed object.

```
λ:  now
         let β be τ;
         .
         .
         .
         thus $\square_{\alpha\leftarrow\beta}$ Justification
    end;
─────────────────────────────────────────────
      for α being τ holds $\square$ by λ
```

where α and β are permitted to be identical object identifiers. The formula $\square_{\alpha\leftarrow\beta}$ must not contain any free occurrences of any object identifier introduced inside the `now` other than β.

Remember that any formula visible at line λ which has β free will not be *properly* visible inside the reasoning. The 'value' of the object identifier β defined outside of the reasoning is not arbitrary (many things could have already been stated about it), and so any formula with a free β from outside the proof is actually referring to a different object than the β inside the proof. (Go back now and reread the definition of visibility).

(ex33.mse) So for example, the following inference is correct

```
environ
    given Fred being PERSON;
    A: Knight[Fred];
begin
         == We prove that for all persons, Fred is a knight.
         == (Do not worry that it does not make much sense.)

    1: now
        let p be PERSON;
        thus Knight[Fred] by A;                         == RE
    end;

    for p being PERSON holds Knight[Fred] by 1          == UI
```

(ex35.mse) while this one is not:

```
environ
    given Fred being PERSON;
    A: Knight[Fred];
begin
            == This says the same as in the previous example.
            == But it tries to prove that all persons are knights!
    1: now
        let Fred be PERSON; == You can do it.

            == The following does not work as Fred mentioned in axiom A
            == is not the same Fred as inside this reasoning.

        thus Knight[Fred] by A;
****************************99
**
**  99 Your inference is not accepted by the checker.
**
    end;

    for p being PERSON holds Knight[Fred] by 1;                 == UI
**********************************************99
**
**  99 Your inference is not accepted by the checker.
**

        == Note. The reasoning at 1 was incorrect, still we can make
        == a reference to it.
        == The above does not work, but the step below does.

    for Fred being PERSON holds Knight[Fred] by 1;              == UI
```

We can now complete the example where we wanted to instantiate an inner quantifier. Note how we do renaming of the quantified object from `y` in the `now`, to `yy` in the concluded formula 2. (ex34.mse)

```
environ
    given Fred being PERSON;
    given Hello being STATEMENT;

    KnightSpeak: for y being PERSON holds
                    for s being STATEMENT holds
                       Knight[y] & Tells[y,s] implies True[s];

begin
         == We want to substitute Hello for s.
  1: now
       let y be PERSON;
         1: for s being STATEMENT holds
              Knight[y] & Tells[y,s] implies True[s] by KnightSpeak; == UE
       thus Knight[y] & Tells[y,Hello] implies True[Hello] by 1;     == UE
  end;

2: for yy being PERSON holds
        Knight[yy] & Tells[yy,Hello] implies True[Hello] by 1        == UI
```

7.8 Existential Elimination, ExE

Our last inference rule is used to convert an existential statement about some object, into a statement about a specific object. The idea is that if we have

$$\texttt{ex}\ \alpha\ \texttt{being}\ \tau\ \texttt{st}\ \boxed{}$$

then $\boxed{}_{\alpha\leftarrow\beta}$ will hold for some specific object β of sort τ. But since we do not know what that object actually is, the name β that we give it should be different from any other visible name. Otherwise, unexpectedly, we can give a new meaning to a name that we have been using. The inference rule looks like

$$\frac{\texttt{ex}\ \alpha\ \texttt{being}\ \tau\ \texttt{st}\ \boxed{}}{\texttt{consider}\ \beta\ \texttt{being}\ \tau\ \texttt{such that}\ \lambda\texttt{:}\ \boxed{}_{\alpha\leftarrow\beta}}$$

Where the formula $\boxed{}$ must not contain any free β.

The `consider` statement introduces a new object with identifier β and sort τ. This will hide all prior instances of β and any prior formulas with a free β cannot be used in the current scope as they contain the now hidden previous β.

(ex37.mse) Here is a complicated example illustrating this inference rule.

```
environ
    given Hello being STATEMENT;
    given Fred being PERSON;
                == Hello is a true statement.
Fact1: True[Hello];
                == Fred always lies.
Fact2: for s being STATEMENT holds Tells[Fred,s] implies not True[s];
                == Every person is either a Knight or a Knave, but not both.
Opp: for a being PERSON holds Knight[a] iff not Knave[a];
                == A person is a knight if and only if every statement
                == they say is true.
Veracity: for a being PERSON holds
             Knight[a] iff
                (for s being STATEMENT holds Tells[a,s] implies True[s]);
                == If at least one of the things you say is false
                == then everything you say is false.
Consistency: for a being PERSON holds
              (ex s being STATEMENT st Tells[a,s] & not True[s]) implies
               (for s being STATEMENT holds Tells[a,s] implies not True[s]);
                == Every person says at least one thing.
Verbosity: for p being PERSON holds ex s being STATEMENT st Tells[p,s];
                == Some person says everything.
BlabberMouth: ex p being PERSON st for s being STATEMENT holds Tells[p,s];

begin

                == We demonstrate that there is some person who says Hello.
                == Use the fact that blabber mouth exists.
```

```
consider blabber being PERSON such that
  1: for s being STATEMENT holds Tells[blabber,s]
                                      by BlabberMouth;         == ExE
            == Combine this with the fact that Hello is a statement.
  2: Tells[blabber, Hello] by 1;                               == UE
            == Now existentially generalize
ex p being PERSON st Tells[p,Hello] by 2;                      == ExI

    == We prove that there exists some person p such that if they
    == are a knight then all statements are true.
    == This example should demonstrate to you that the scope of
    == an existential quantifier better not be a conditional,
    == if it is possible for the antecedent to be false for some
    == object.

consider blabber being PERSON such that
  1: for s being STATEMENT holds Tells[blabber,s]
                                      by BlabberMouth;         == ExE
  2': now
         assume A: Knight[blabber];
                3: for s being STATEMENT holds
                      Tells[blabber,s] implies True[s]
                                      by Veracity, A;           == UE, EqE
                4: now
                    let s be STATEMENT;
                       5: Tells[blabber,s] by 1;                == UE
                    thus True[s] by 3, 5;                       == UE, IE
                  end;
         thus for s being STATEMENT holds True[s] by 4;         == UI
     end;

  2: Knight[blabber] implies
           (for s being STATEMENT holds True[s]) by 2';         == II

ex p being PERSON st
    (Knight[p] implies (for s being STATEMENT holds True[s]))
                                               by 2;            == ExI

    == Note that we can reference 2' directly.

ex p being PERSON st
    (Knight[p] implies (for s being STATEMENT holds True[s]))
                                               by 2';           == ExI
```

In the example above we have indicated the lines where two basic rules of inference have been used, in the indicated order. Although, we need not doing it to satisfy the MIZAR checker, sometimes it helps to clarify things for ourselves.

7.9 De Morgan's Laws for Quantifiers

As with derived inference rules for propositional calculus, there are derived inference rules for quantifiers. Here we mention only the so called De Morgan's Laws[4] and attach the corresponding argument using only the basic rules of inference.

1.

$$\frac{\texttt{not (for } \alpha \texttt{ being } \tau \texttt{ holds } \boxed{}\texttt{)}}{\texttt{ex } \alpha \texttt{ being } \tau \texttt{ st not } \boxed{}}$$

(ex38.mse)

```
environ
        A1: not (for a being T holds PHI[a]);
begin

1: now
    assume
     step1: not (ex a being T st not PHI[a]);

     step2: now
             let x be T;

              step3': now assume
                        step3: not PHI[x];
                        step4: ex a being T st not PHI[a] by step3;  == ExI
                       thus contradiction by step1, step4;           == ContrI
                      end;

             thus PHI[x] by step3';                                  == PbyC
            end; == We have just shown: for a being T holds PHI[a].

    thus contradiction by A1, step2;                                 == ContrI
   end;

ex a being T st not PHI[a] by 1                                      == PbyC
```

2.

$$\frac{\texttt{not (ex } \alpha \texttt{ being } \tau \texttt{ st } \boxed{}\texttt{)}}{\texttt{for } \alpha \texttt{ being } \tau \texttt{ holds not } \boxed{}}$$

(ex39.mse)

```
environ
        A1: not (ex a being T st PHI[a]);
begin
```

[4]See 6.3.11 for the De Morgan's laws in their quantifier-free version.

```
 1: now
      let x be T;
        step0: now
                 assume step1: PHI[x];
                        step2: ex a being T st  PHI[a] by step1; == ExI
                 thus contradiction by A1, step2;                == ContrI
               end;
      thus not PHI[x] by step0                                   == NI
     end;

for a being T holds not PHI[a] by 1;                             == UI
```

3.

$$\frac{\texttt{for } \alpha \texttt{ being } \tau \texttt{ holds not } \Box}{\texttt{not (ex } \alpha \texttt{ being } \tau \texttt{ st } \Box\texttt{)}}$$

(ex40.mse)

```
environ
        A1: for a being T holds not PHI[a];
begin
 1: now
      assume step1: ex a being T st PHI[a];
                    consider x being T such that
             step2: PHI[x] by step1;                == ExE
             step3: not PHI[x] by A1;               == UE
      thus contradiction by step2, step3;           == ContrI
 end;

not (ex a being T st PHI[a]) by 1                   == PbyC
```

4.

$$\frac{\texttt{ex } \alpha \texttt{ being } \tau \texttt{ st not } \Box}{\texttt{not (for } \alpha \texttt{ being } \tau \texttt{ holds } \Box\texttt{)}}$$

(ex41.mse)

```
environ
        A1: ex a being T st not PHI[a];
begin
 1: now
      assume step1: for a being T holds PHI[a];
                    consider x being T such that
             step2: not PHI[x] by A1;               == ExE
             step3: PHI[x] by step1;                == UE
      thus contradiction by step2, step3;           == ContrI
 end;

not (for a being T holds PHI[a]) by 1               == PbyC
```

8 Proofs

The inference rules that we have covered so far suffice to prove whatever is provable in a given environment. (The above claim requires a proof but we do not discuss it here. It falls into the area of metalogic—that is knowledge about logic. Our interest is here about practical aspects of proving.)

Any justification of a formula in MIZAR may take the form of a *proof.* In essence, proofs are only a syntactic variation of recording the application of inference rules, which offers some convenient short cuts in conducting arguments. The syntax of proofs is similar to the syntax of reasonings.

CompactStatement ::= [`then`] *Proposition* *SimpleJustification* |
Proposition *Proof*

Proof ::= `proof` *Reasoning* `end`

8.1 `II`, `NI`, `PbyC`, `UI` Rules Revisited

Using examples, we show how the proof syntax is used for recording the `II`, `NI`, `PbyC`, and `UI` rules.

8.1.1 Proof of an Implication

The proof form of the `II` rule is identical to the `now` version of the `II` rule, see 6.1. Indeed, what goes between `now` and `end` now goes between `proof` and `end`, and altogether save a label and a `by` reference.

$\boxed{0\,|\quad\quad}$ `implies` $\boxed{1\,|\quad\quad}$ `proof`
    `assume A:` $\boxed{0\,|\quad\quad}$
    .
    .
    .
    `thus` $\boxed{1\,|\quad\quad}$ *Justification*
`end`

Let us look at the proof form of the example from p. 80. (Note the place where we announce (ex43.mse) the rule name.)

```
environ
      given x being PERSON;
 Ax1: P[x] implies Q[x];
 Ax2: Q[x] implies R[x];
begin

    P[x] implies R[x] proof            == Rule: II
        assume A: P[x];
              1a: Q[x] by A, Ax1;      == Rule: IE
        thus R[x] by 1a, Ax2           == Rule: IE
    end
```

Whatever we can prove with a `proof` we can also justify using the `now` reasoning. There is a psychological difference though. To use a proof we have to state first what we are proving, while in case of the `now` reasoning we can do it later, if we wish, when we state explicitly what has been proven. (Certainly, in both cases the Mizar checker will tell us whether our statements are justified.)

For example, let us prove this theorem about a `PERSON` called `x`.

```
( (P[x] implies Q[x]) & (Q[x] implies R[x]) & P[x] ) implies R[x]
```

(ex44.mse) We do it both ways, presenting the proof first.

```
environ
    given x being PERSON;
begin

== The proof version

    ( (P[x] implies Q[x]) & (Q[x] implies R[x]) & P[x] ) implies R[x]
    proof         == Rule: II
        assume A: (P[x] implies Q[x]) & (Q[x] implies R[x]) & P[x];
               1: P[x] by A;                 == CE
               2: P[x] implies Q[x] by A;    == CE
               3: Q[x] by 1, 2;              == IE
               4: Q[x] implies R[x] by A;    == CE
        thus R[x] by 3,4;                    == IE
    end;

== The now version

 1: now
        assume A: (P[x] implies Q[x]) & (Q[x] implies R[x]) & P[x];
               1: P[x] by A;                 == CE
               2: P[x] implies Q[x] by A;    == CE
               3: Q[x] by 1, 2;              == IE
               4: Q[x] implies R[x] by A;    == CE
        thus R[x] by 3,4;                    == IE
    end;

    ( (P[x] implies Q[x]) & (Q[x] implies R[x]) & P[x] ) implies R[x]
                                              by 1;     == II
```

The justification of the main proposition is in both cases via hypothetical reasoning (*Implication Introduction*).

8.1.2 Proof of a Negation

Instead of applying the `NI` rule as presented on page 82 we may present the following proof:

```
not $\boxed{0}\boxed{\phantom{xxx}}$ proof
    assume Contra:  $\boxed{0}\boxed{\phantom{xxx}}$;
    .
    .
    .
    thus contradiction $\textit{Justification}$
end
```

(ex45.mse) Compare the `proof` and the `now` versions of the example from page 82.

```
environ
    given glass being CONTAINER;
    given juice being DRINK;
    A1: Full[glass] implies Fresh[juice];
    A2: not Fresh[juice];
begin       == We show that: not Full[glass]

== The proof version

    not Full[glass] proof                 == NI
        assume A: Full[glass];
               1: Fresh[juice] by A, A1;  == IE
        thus contradiction by 1,A2;       == ContrI
    end;

== The now version

   S: now
        assume A: Full[glass];
               1: Fresh[juice] by A, A1;  == IE
        thus contradiction by 1,A2;       == ContrI
    end;
    not Full[glass] by S                  == NI
```

8.1.3 Proof by Contradiction

There is also a full correspondence between the `PbyC` rule and the proof version of reasoning by contradiction.

```
$\boxed{0}\boxed{\phantom{xxx}}$ proof
    assume Contra:  not $\boxed{0}\boxed{\phantom{xxx}}$;
    .
    .
    .
    thus contradiction $\textit{Justification}$
    end
```

(ex46.mse) The example from page 82 takes the following proof form.

```
environ
    given glass being CONTAINER;
begin

    (not Full[glass] implies Full[glass]) implies Full[glass]
    proof                                                      == II
        assume a: not Full[glass] implies Full[glass];
               S: now                                          == NI
                   assume b: not Full[glass];
                          1: Full[glass] by a,b;               == MP
                   thus contradiction by 1,b;                  == ContrI
                  end;
               T: not not Full[glass] by S;                    == NI
        thus Full[glass] by T                                  == NE
    end;
```

8.1.4 Universal Introduction

The following proof schema is fully analogous to the UI rule.

```
$\lambda$:  for $\alpha$ being $\tau$ holds $\boxed{\phantom{xx}}$ proof
     let $\beta$ be $\tau$;
     .
     .
     .
     thus $\boxed{\phantom{xx}}_{\alpha \leftarrow \beta}$ *Justification*
    end
```

(ex47.mse) And now, employing a proof, the example from page 104 looks as follows:

```
environ
    given Fred being PERSON;
    given Hello being STATEMENT;

    KnightSpeak: for y being PERSON holds
                    for s being STATEMENT holds
                       Knight[y] & Tells[y,s] implies True[s];

begin

for y being PERSON holds
        Knight[y] & Tells[y,Hello] implies True[Hello]              == UI
   proof
       let y be PERSON;
         1: for s being STATEMENT holds
              Knight[y] & Tells[y,s] implies True[s] by KnightSpeak; == UE
       thus Knight[y] & Tells[y,Hello] implies True[Hello] by 1;     == UE
```

```
end
```

8.2 Proof Structures

We have seen above how the hypothetical reasoning (`II`), Negation Introduction `NI`, and Universal Introduction `UI` rules take on the a shape of a proof. Thus we know how to write a proof for a formula with an implication, negation, or a universal quantifier as the main connective. We also know how to prove an arbitrary formula by contradiction. Mizar allows us to write a proof for an arbitrary formula, even if its structure is suitable to use a *SimpleJustification* on a basis of some rule of inference.

8.2.1 Conjunction

In order to prove a conjunction (directly) we have to demonstrate the truth of all its conjuncts. One can always use the `CI` rule. But one can employ the following proof structure.

```
|0|    | & |1|    | & .  .  . & |n|    | proof
     .
     thus |0|    |   Justification ;
     .
     thus |1|    |   Justification ;
     .
     .
     .
     thus |n|    |   Justification
end
```

Usually, there is no point in employing this proof structure—applying the `CI` rule results in a shorter text.

8.2.2 Implication

The hypothetical reasoning forms the direct proof structure for a conditional formula. At this point we would like to mention the idea of **indirect proof**. You may note the affinity of the `II` rule and the *Modus Ponens* rule. One can wonder whether there is a similar proof structure corresponding to the *Modus Tollens* rule. We would like to permit the following, *indirect*, structure for a proof of a conditional:

```
|0|    | implies |1|    | proof
     assume not |1|    |;
     .
     .
     .
     thus not |0|    |   Justification
end
```

From the *Modus Tollens* rule we can feel that such a structure is right. And this is the case. Unfortunately, Mizar **does not recognize it** as a proper structure for proving a conditional. Too bad! But we can always find our way around the problem and do the indirect proof with the `now` reasoning as follows:

```
    A: now
        assume not $\boxed{1\quad}$;
        .
        .
        .
        thus not $\boxed{0\quad}$    *Justification*
    end;
B: not $\boxed{1\quad}$ implies not $\boxed{0\quad}$ by A;
$\boxed{0\quad}$ implies $\boxed{1\quad}$ by B == Reverse Implication
    == MIZAR will accept the following as well:
$\boxed{0\quad}$ implies $\boxed{1\quad}$ by A == No single rule suffices
```

All proofs that are not by contradiction and are not indirect, are called direct proofs.

8.2.3 Disjunction

The direct proof of a single disjunction looks quite 'indirect', although it is not. We already know (from the example on page 92) which conditional formula is equivalent to a given disjunction. MIZAR permits the following as a proof of a disjunction.

```
$\boxed{0\quad}$ or $\boxed{1\quad}$ proof
    assume not $\boxed{0\quad}$;
    .
    .
    .
    thus $\boxed{1\quad}$    *Justification*
end
```

Unfortunately, we cannot start with the negation of the second disjunct. But always we can call on the `now` reasoning for help.

8.2.4 Equivalence

The direct proof of an equivalence must take the following form:

```
$\boxed{0\quad}$ iff $\boxed{1\quad}$ proof
    . . .
    thus $\boxed{0\quad}$ implies $\boxed{1\quad}$    *Justification* ; .
    .
    .
    thus $\boxed{1\quad}$ implies $\boxed{0\quad}$    *Justification* ;
end
```

Unfortunately for MIZAR, the order of these `thus`'es is important.

8.3 Omitting Proofs

The correctness of a formula may be established by *SimpleJustification* without writing an explicit `proof` for it. Whenever we get the MIZAR processor to accept

$\boxed{}\!\boxed{}$ `by` *References*

then we can also justify $\boxed{}\!\boxed{}$ using the following proof structure:

$\boxed{}\!\boxed{}$ `proof`
        .
        .
        .
        `thus` $\boxed{}\!\boxed{}$ `by` *References*
`end`

Since the latter is longer we will prefer the former way of justification. Note how the above proof structure is similar to the one we discussed for conjunction in 8.2.1. Indeed, any formula can be seen as a conjunction with exactly one conjunct, no matter how strange it sounds.

8.4 Incorporating proofs

A proof nested within another proof can be incorporated directly into the encompassing (ex24.mse) proof. We start with the example, which is a variation of one of the proofs from page 93.

```
environ
    given S being STUDENT;
    given M being Money;
begin
    (Need[M] implies Work[S] & Save[M])
        implies
    (Need[M] implies Work[S]) & (not Save[M] implies not Need[M])
    proof                                                          == II
        assume A:( Need[M] implies Work[S] & Save[M] );

== At this point we have to prove the consequent of the original formula.
== Since it is a conjunction, we do it in two separate thus'es.

        thus Need[M] implies Work[S] proof                         == II
               assume A1: Need[M];
                      1a: Work[S] & Save[M] by A, A1;              == IE
               thus Work[S] by 1a;                                 == CE
           end;

        thus not Save[M] implies not Need[M] proof                 == II
              assume A1: not Save[M];
                     2a: Need[M] implies Save[M] proof             == II
                          assume A2: Need[M];
                                 2b: Work[S] & Save[M] by A, A2;   == IE
                          thus Save[M] by 2b;                      == CE
                       end;
              thus not Need[M] by A1, 2a;                          == MT
           end;
    end;
```

This proof can still be refined by incorporating the last part of the conclusion into the main (ex22.mse) proof.

```
environ
    given S being STUDENT;
    given M being Money;
begin
    (Need[M] implies Work[S] & Save[M])
        implies
    (Need[M] implies Work[S]) & (not Save[M] implies not Need[M])
    proof                                                          == II
        assume A:( Need[M] implies Work[S] & Save[M] );

== At this point we have to prove the consequent of the original formula.
== It is a conjunction. The first conjunct is concluded by a thus.
```

```
        thus Need[M] implies Work[S] proof                              == II
               assume A1: Need[M];
                      1a: Work[S] & Save[M] by A, A1;                   == IE
               thus Work[S] by 1a;                                      == CE
           end;

== At this point the second conjunct has to be concluded.
== Since it is an implication we incorporate the proof of the implication
== into the main proof.
== We are really doing the implication introduction from here on.

        assume A1: not Save[M];

== What remains to be proven at this point is: not Need[M]

               2a: now                                                  == II
                    assume A2: Need[M];
                           2b: Work[S] & Save[M] by A, A2;              == IE
                    thus Save[M] by 2b;                                 == CE
               end;

        thus not Need[M] by A1, 2a;                                     == MT
    end;
```

The key point to understanding all this is the question: What remains to be proven in a proof? It is a formula. Let us introduce a name for it. That which still remains to be proven in a proof we call the current *thesis*. Initially, right after the `proof`, the current thesis is the original formula that we are proving. Each assumption (`assume`) and conclusion (`thus`) that we make affects the thesis. The precise explanation of *What is the current thesis?* is pretty obscure. Therefore, we will rely more on your intuition. Assume we are at the beginning of a proof. The current thesis is the original formula. If we go through an assumption `assume` then it looks like we are applying hypothetical reasoning and the current thesis better be an implication. After the assumption the current thesis is the consequent of the previous thesis. And now, if the current thesis is an implication we may expect another assumption; if not we have to conclude (`thus`) whatever remains to be proven. At any point in a proof you can assume the negation of the current thesis, which means that you are switching to a proof by contradiction.

Mizar has been equipped with a special contraption to talk about the current thesis.

8.5 `thesis`

We know the fundamental proof structures for propositional formulas. We know also we can use a proof by contradiction. For a conditional formula we can present an indirect reasoning. We know how to structure a proof to get a universally quantified formula.

The previous example will be used to introduce the special formula named `thesis`. In a proof, `thesis` denotes the current thesis, which is the formula that remains to be concluded in order to complete the innermost proof. The proof that can always be omitted (page 116) can be then rewritten (although there is no need to write it in the first place) as:

```
$\boxed{\phantom{0}}\boxed{\phantom{xxx}}$ proof
    .
    .
    .
    thus thesis by  *References*
end
```

Outside of a proof `thesis` must not be used as it is meaningless. It is quite natural to use `thesis` in hypothetical reasoning.

```
$\boxed{0}\boxed{\phantom{xxx}}$ implies $\boxed{1}\boxed{\phantom{xxx}}$ proof
    .         == thesis  is here $\boxed{0}\boxed{\phantom{xxx}}$ implies $\boxed{1}\boxed{\phantom{xxx}}$
    assume $\boxed{0}\boxed{\phantom{xxx}}$;
    .         == thesis  is here $\boxed{1}\boxed{\phantom{xxx}}$
    .
    .
    thus thesis  *Justification*
    .         == thesis  is here not contradiction
    .         == which does not have to be concluded by a thus
end
```

It is quite convenient to use `thesis` in a proof by contradiction, it may shorten our proofs.

```
$\boxed{0}\boxed{\phantom{xxx}}$ proof
    .         == thesis  is here $\boxed{0}\boxed{\phantom{xxx}}$
    assume not thesis;
    .         == thesis  is here contradiction
    .
    .
    thus thesis  *Justification*
    .         == thesis  is here not contradiction
    .         == which does not have to be concluded by a thus
end
```

It is interesting to see the behaviour of `thesis` when doing universal introduction. Let (ex42.mse) us look at an example.

```
environ
   1: for p being Person holds Knight[p] implies Wise[p];
   2: for p being Person holds Wise[p] implies Rich[p];
begin

   for p being Person holds Knight[p] implies Rich[p] proof      == UI
      let P be Person;
             == thesis   <=>   Knight[P] implies Rich[P]
      thus thesis proof                                          == II
                  == thesis   <=>   Knight[P] implies Rich[P]
              assume a: Knight[P];
                  == thesis   <=>   Rich[P]
                     b: Wise[P] by 1, a;                         == UE, IE
                     c: Rich[P] by 2, b;                         == UE, IE
              thus thesis by c                                   == RE
                  == thesis   <=>  not contradiction
           end;   == thesis   <=>  not contradiction
   end;
           == The inner proof can be shortened.

   for p being Person holds Knight[p] implies Rich[p] proof      == UI
      let P be Person;
             == thesis   <=>   Knight[P] implies Rich[P]
      thus thesis proof                                          == II
                  == thesis   <=>   Knight[P] implies Rich[P]
              assume a: Knight[P];
                  == thesis   <=>   Rich[P]
                     b: Wise[P] by 1, a;                         == UE, IE
              thus thesis by 2, b;                               == UE, IE
                  == thesis   <=>  not contradiction
           end;   == thesis   <=>  not contradiction
   end;
           == The inner proof can be incorporated entirely.

   for p being Person holds Knight[p] implies Rich[p] proof      == UI
      let P be Person;
             == thesis   <=>   Knight[P] implies Rich[P]
             == Since the thesis is a conditional we can use
             == hypothetical reasoning to get it.
      assume a: Knight[P];                                       == II
             == thesis   <=>   Rich[P]
        b: Wise[P] by 1, a;                                      == UE, IE
      thus thesis by 2, b
                  == thesis   <=>  not contradiction
   end;
```

8.6 MIZAR Problems on Knights and Knaves

8.6.1 Puzzle 1

(kandk02.mse)

```
environ

given A, B being PERSON;

Opp: for a being PERSON holds Knight[a] iff not Knave[a];

== These two statements are a translation of the sentence, said by A,
== " I am a knave or B is a knight"

A1:  Knight[A] implies ( Knave[A] or Knight[B] );
A1': Knave[A] implies not ( Knave[A] or Knight[B] );

begin

== The question is what are A and B?

== First we argue that A must be a knight.

C1: Knight[A]
    proof
        assume Contra: not Knight[A];
                   C1: Knave[A] by Contra, Opp;
                   C2: not ( Knave[A] or Knight[B] ) by C1, A1';
                   C3: not Knave[A] by C2;
                   C4: Knight[A] by C3, Opp;
        thus contradiction by C4, Contra;
    end;

== Now we determine what B is.

D1: Knight[B]
    proof
        S1: Knave[A] or Knight[B] by C1, A1;
        S2: not Knave[A] by C1, Opp;
        thus thesis by S1, S2;
    end;
```

8.6.2 Puzzle 2

(kandk03.mse)

```
environ

given A, B, C being PERSON;

Opp: for a being PERSON holds Knight[a] iff not Knave[a];

== These two are a translation of the sentence, said by A,
== "All of us are knaves."
A1:  Knight[A] implies ( Knave[A] & Knave[B] & Knave[C] );
A1':  Knave[A] implies not ( Knave[A] & Knave[B] & Knave[C] );


== These two are a translation of the sentence, said by B,
== "Exactly one of us is a knight."
B1: Knight[B] implies ( (Knight[A] & Knave[B] & Knave[C]) or
   (Knave[A] & Knight[B] & Knave[C]) or
   (Knave[A] & Knave[B] & Knight[C]) );
B1': Knave[B] implies not ( (Knight[A] & Knave[B] & Knave[C]) or
    (Knave[A] & Knight[B] & Knave[C]) or
    (Knave[A] & Knave[B] & Knight[C]) );

begin

== The question is what are A, B and C?

== First we argue that A must be a knave.
A2: Knave[A]
    proof
        assume Contra: not Knave[A];
                    1: Knight[A] by Contra, Opp;
                    2: Knave[A] by 1, A1;
        thus contradiction by 2, Contra;
    end;

== A useful sentence:
B2: ( (Knight[A] & Knave[B] & Knave[C]) or
      (Knave[A] & Knight[B] & Knave[C]) or
      (Knave[A] & Knave[B] & Knight[C]) )
    implies not Knave[B] by B1';

== Then we argue that if C is a knight then B must be a knight.
C2: Knight[C] implies Knight[B]
    proof
        assume S1: Knight[C];
               S2: Knave[A] & Knight[C] by S1, A2;
               S3: not Knave[B]
                   proof
```

```
                       assume Contra: Knave[B];
                                  31: Knave[A] & Knave[B] & Knight[C]
                                      by S2, Contra;
                                  32: not Knave[B] by B2, 31;
                       thus contradiction by Contra, 32;
                   end;
        thus Knight[B] by Opp, S3;
    end;

== Then we argue that if C is a knave then B must still be a knight.
C3: Knave[C] implies Knight[B]
    proof
        assume S1: Knave[C];
        thus Knight[B]
             proof
                 assume Contra: not Knight[B];
                            1: Knave[B] by Opp, Contra;
                            2: Knave[A] & Knave[B] & Knave[C] by A2, S1, 1;
                            3: not Knave[A] by A1', 2;
                 thus contradiction by A2, 3;
             end;
    end;

== A tautology:
C4: Knight[C] or not Knight[C];
C5: Knight[C] or Knave[C] by C4, Opp;

== No matter what C is, B is a Knight.
B3: Knight[B] by C2, C3, C5;

C6: Knave[C] by A1, A1', B1, B1', B3, C5;

Conclusion: Knave[A] & Knight[B] & Knave[C] by A2, B3, C6;
```

8.6.3 A theorem about Knights and Knaves

(kandk04.mse)

```
environ

== Opp says that every person is either a Knight or a Knave, but
== not both
Opp: for a being PERSON holds Knight[a] iff not Knave[a];

== Veracity says that a person is a knight if and only if every statement
== they say is true.
Veracity: for a being PERSON holds
        Knight[a] iff ( for s being STATEMENT holds
        Tells[a,s] implies True[s] )

begin

== Theorem1 says that if some person, p, is a knave then there is at
== least one statement s, uttered by p, that is not true
== But it does not say that every statement of p is false. Can this
== latter assertion be proven with the above environment?

Theorem1: for p being PERSON holds
        Knave[p] implies ( ex s being STATEMENT st
        Tells[p,s] & not True[s] )
    proof
        let p be PERSON;

        == We have to prove the implication.
        assume A1: Knave[p];

               == Now we have to prove the consequent.
               C0: not Knight[p] by A1, Opp;

               C1: Knight[p] iff ( for s being STATEMENT holds
                   Tells[p,s] implies True[s] ) by Veracity;

               C2: not ( for s being STATEMENT holds
                   Tells[p,s] implies True[s] ) by C1, C0;

               C3: ex s being STATEMENT st not
                   ( Tells[p,s] implies True[s] ) by C2;

               consider t being STATEMENT such that
               C4: not ( Tells[p,t] implies True[t] ) by C3;

               C5: Tells[p,t] & not True[t] by C4;

        thus thesis by C5;
    end;
```

8.6.4 Three more theorems

(kandk05.mse)

```
environ

given Hello being STATEMENT;
given Fred being PERSON;

== Hello is a true statement.
Fact1: True[Hello];

== Fred always lies.
Fact2: for s being STATEMENT holds Tells[Fred,s] implies not True[s];

== Opp says that every person is either a Knight or a Knave, but
== not both.

Opp: for a being PERSON holds Knight[a] iff not Knave[a];

== Veracity says that a person is a knight if and only if every statement
== they say is true

Veracity: for a being PERSON holds
        Knight[a] iff ( for s being STATEMENT holds
        Tells[a,s] implies True[s] );

== Consistency says that if at least one of the things a person says is
== false then everything they say is false.

Consistency: for a being PERSON holds (
        (ex s being STATEMENT st Tells[a,s] & not True[s]) implies
        (for s being STATEMENT holds Tells[a,s] implies not True[s]) );

begin

== Theorem2 says that Fred cannot say hello.
Theorem2: not Tells[Fred,Hello] proof
      == Specialize Fact2 to the statement Hello.
      C0: Tells[Fred, Hello] implies not True[Hello] by Fact2;
    thus thesis by C0, Fact1;
end;

== Theorem3 says that if Fred says nothing at all, then he is a knight!
Theorem3: (for s being STATEMENT holds not Tells[Fred,s])
    implies Knight[Fred] proof
        assume A1: (for s being STATEMENT holds not Tells[Fred,s]);
               C0: for s being STATEMENT holds Tells[Fred,s] implies True[s]
                     proof
                         let s be STATEMENT;
                         C1: not Tells[Fred,s] by A1;
```

```
                        thus thesis by C1;
                    end;
                C2: Knight[Fred] iff ( for s being STATEMENT holds
                    Tells[Fred,s] implies True[s] ) by Veracity;
            thus thesis by C0, C2;
        end;

== Theorem4 says that if some person, a, is a knave then every
== statement s uttered by a is false.

Theorem4: for p being PERSON holds
        Knave[p] implies ( for s being STATEMENT holds
        Tells[p,s] implies not True[s] )
    proof
        let p be PERSON;

        == Now we have to prove the implication.
        assume A1: Knave[p];

               == Now we have to prove the consequent.
               C0: not Knight[p] by A1, Opp;

               C1: Knight[p] iff ( for s being STATEMENT holds
                   Tells[p,s] implies True[s] ) by Veracity;

               C2: not ( for s being STATEMENT holds
                   Tells[p,s] implies True[s] ) by C1, C0;

               C3: ex s being STATEMENT st not
                   ( Tells[p,s] implies True[s] ) by C2;

               consider t being STATEMENT such that
               C4: not ( Tells[p,t] implies True[t] ) by C3;

               C5: Tells[p,t] & not True[t] by C4;

               C6: ex s being STATEMENT st Tells[p,s] & not True[s] by C5;

               C7: (ex s being STATEMENT st Tells[p,s] & not True[s]) implies
               (for s being STATEMENT holds Tells[p,s] implies not True[s])
                  by Consistency;

        thus thesis by C6,C7;
    end;
```

8.6.5 And even more theorems

(kandk06.mse)

```
== Here is the intended interpretation of the predicates Tells, Likes, and
== Conspire.
== Tells[Person, Statement] - Tells[a,s] means person a tells statement s.
== Likes[Person, Person] - Likes[a,b] means person a likes person b.
== Conspire[Person, Person, Person] - Conspire[x,y,z] means both x and y
==    do not like z.

environ
        == These reserve statements save us the trouble of specifying
        == the sort of the variables being quantified.
        reserve x, y, z for Person;
        reserve r, s, t for Statement;

        given A, B, C, D, E, F, G being Person;
        given Lie, Promise, Blabla, Insult, Prayer being Statement;

a1:     for x holds Knave[x] or Knight[x] or Normal[x];
a1':    for x holds Knave[x] implies not (Knight[x] or Normal[x]);
a1'':   for x holds Knight[x] implies not (Knave[x] or Normal[x]);
a1''':  for x holds Normal[x] implies not (Knight[x] or Knave[x]);

a2:     for x, y holds Likes[x, y] & Knight[x] implies Knave[y];
a2':    for x, y holds Likes[x, y] & Knave[x] implies not Normal[y];
a2'':   for x, y holds not Likes[x, y] & Normal[x] implies Knight[y];

a3:     for x, y, z holds Conspire[x, y, z] implies
                        not (Likes[x, z] or Likes[y, z]);
a3':    for x, y, z holds
                not Likes[x, z] & not Likes[y, z] &
                not (Likes[z, x] or Likes[z, y])
                                implies Conspire[x, y, z];

a4:     for x, s holds Knight[x] & Tells[x, s] implies True[s] & s <> Insult;
a4':    for x, s holds Knave[x] & Tells[x, s] implies False[s] or s = Blabla;
a4'':   for x holds Normal[x] implies Tells[x, Promise];

a5:     for x being Person holds Likes[x, D];
a5':    not False[Prayer];
a5'':   Prayer <> Blabla & Promise <> Blabla;
a5''':  for x holds Knave[x] implies not Likes[E, x];

a6:     Likes[F, G] & Likes[G, F];
a6':    Normal[F];


begin
```

```
Exercise1:

not Knave[G] proof
    1: Normal[F] by a6';
    2: Likes[G, F] & Knave[G] implies not Normal[F] by a2';
    3: not ( Likes[G, F] & Knave[G] ) by 1, 2;
    4: not Likes[G, F] or not Knave[G] by 3;
    thus not Knave[G] by a6, 4;
end;


Exercise2:

not Conspire[F, F, G] proof
    1: Likes[F, G] by a6;
    2: Conspire[F, F, G] implies not (Likes[F, G] or Likes[F, G]) by a3;
    thus thesis by 1,2;
end;


Exercise3:

not Conspire[A, E, D] proof
    1: Likes[A, D] by a5;
    2: Conspire[A, E, D] implies not (Likes[A, D] or Likes[E, D]) by a3;
    thus thesis by 1,2;
end;


Exercise4:

not Normal[C] & Tells[C, Promise]
  implies True[Promise] or False[Promise] proof
    assume A: not Normal[C] & Tells[C, Promise];
           1: Knave[C] or Knight[C] by A, a1;
           2: Knight[C] implies True[Promise] by A, a4;
           3: Knave[C] implies False[Promise] by A, a4', a5'';
    thus thesis by 1,2,3;
end;


Exercise5:

(Likes[E, A] iff Likes[E, B]) & Knave[B] & Normal[E] & Tells[A, Promise]
  implies True[Promise]   proof
    assume A: (Likes[E, A] iff Likes[E, B]) & Knave[B] & Normal[E] &
               Tells[A, Promise];

    1: Knave[B] implies not Likes[E, B] by a5''';
    2: not Likes[E, B] by A, 1;
    3: not Likes[E, A] by A, 2;
    4: Knight[A] by 3, A, a2'';
    5: Knight[A] implies True[Promise] by a4, A, a5'';

    thus thesis by 5, 4;
end;


Exercise6:

Likes[B, A] & Knight[B] & Tells[A, Promise]
  implies False[Promise]  proof
    1: Likes[B, A] & Knight[B] implies Knave[A] by a2;
    2: Knave[A] & Tells[A, Promise] implies False[Promise] by a4', a5'';
    thus thesis by 1,2;
end;


Exercise7:

not Tells[D, Promise] implies Knight[D] proof
    assume A: not Tells[D, Promise];
           1: not Normal[D] by a4'', A;
           2: Likes[E, D] by a5;
           3: Knave[D] implies not Likes[E,D] by a5''';
           4: not Knave[D] by 2, 3;
    thus thesis by 1, 4, a1;
end;


Exercise8:

Knave[B] & Knave[C] & Conspire[A, B, E] & Conspire[A, C, E]
  implies Conspire[B, C, E]       proof
    assume A: Knave[B] & Knave[C] & Conspire[A, B, E] & Conspire[A, C, E];
           1: not Likes[E, B] & not Likes[E, C] by A, a5''';
           2: not Likes[B, E] by a3, A;
           3: not Likes[C, E] by a3, A;
    thus thesis by 1, 2, 3, a3';
end;


Exercise9:

Conspire[D, A, E] & Tells[E, Insult]
  implies not (Normal[D] & Normal[A]) proof
    assume A: Conspire[D, A, E] & Tells[E, Insult];
           1: not Likes[D, E] & not Likes [A, E] by A, a3;
           2: Knight[E] implies Insult <> Insult by A, a4;
           3: not Knight[E] by 2;
           4: not Normal[A] by a2'', 1, 3;
           5: not Normal[D] by a2'', 1, 3;
    thus thesis by 4, 5;
end;


Exercise10:

Tells[A, Prayer] & not Tells[A, Blabla] & not Tells[A, Promise]
  implies Knight[A]       proof
    assume A: Tells[A, Prayer] & not Tells[A, Blabla] & not Tells[A, Promise];
           1: not False[Prayer] & not ( Prayer = Blabla )
               implies not (Knave[A] & Tells[A,Prayer]) by a4';
           2: not Knave[A] or not Tells[A, Prayer] by 1, a5', a5'';
           3: not Knave[A] by 2, A;
           4: not Normal[A] by a4'', A;
    thus thesis by 3, 4, a1;
end;
```

9 Interpretations Revisited

The truth value of tautologies and contradictions does not depend on the current interpretation. However, if a formula is a contingency, we should be able to find an interpretation (ex48.mse) that satisfies the formula, and one that falsifies it. Here is an example:

```
environ == Examples of satisfying and falsifying interpretations

reserve t, x, y, z, w for THING;

    == We will consider three different interpretations of Blob and
    == the universe of discourse (sort THING).

I1: for x, y holds (Blob[x, y] iff x=y);
I1': ex x, y st x <> y;

I2: for x, y holds (Blob[x, y] iff x=y);
I2': not (ex x, y st x <> y);

I3: for x, y holds (Blob[x, y] iff not contradiction);

begin
          == The following is always true, but requires a proof.

Tfor: for x holds x=x     proof let x be THING; thus x=x; end;

          == In Mizar, each sort always contains at least one member.
          == Thus we can always prove the following.

Tex: ex x st x=x          proof consider t; 1: t = t; thus thesis by 1; end;

          == Mizar does not need the above proof.

     consider t;
Tex: ex x st x=x;

    == Statement T0 is true no matter what interpretation we give to Blob,
    == but Mizar is not able to prove it itself.

T0: (ex x st for y holds Blob[x, y]) implies (for y ex x st Blob[x, y])
    proof assume A1: ex x st for y holds Blob[x, y];
          let z be THING;
             consider t such that
          1: for y holds Blob[t, y] by A1;
          2: Blob[t, z] by 1;
       thus  thesis by 2;
    end;

    == Note how in proving T0 we did not make any reference to the
    == interpretation axioms.
    == But the converse of T0 is not always true.
    == Under interpretation I1 it is false.

T1: not ((for y ex x st Blob[x, y]) implies (ex x st for y holds Blob[x, y]))
    proof
       assume A1: (for y ex x st Blob[x, y]) implies
                                    (ex x st for y holds Blob[x, y]);
          == The antecedent is true using interpretation I1

          1:  for y ex x st Blob[x, y] proof
                 let y be THING;
                   2: y=y;
                   3: Blob[y, y] by I1;
                 thus ex x st Blob[x, y] by 3;
              end;
          4: ex x st for y holds Blob[x, y] by 1, A1;
                        == Now get a contradiction
             consider t such that
          5: for y holds Blob[t, y] by 4;
             consider w, z such that
          6: w<>z by I1';
          7: Blob[t, w] & Blob[t, z] by 5;
          8: t=w by 7, I1;
          8': t=z by 7, I1;
          9: z=w by 8, 8';
       thus contradiction by 6, 9;
end;

         == But under interpretation I2 it is true.

T2: (for y ex x st Blob[x, y]) implies (ex x st for y holds Blob[x, y])
    proof
       assume A1: for y ex x st Blob[x, y];
             consider t;
             consider w such that
          2: Blob[w, t] by A1;
          3: for y holds Blob[w, y] proof
                 let y be THING;
                    4: y=t by I2';
                    5: Blob[w, y] by 2, 4;
                 thus thesis by 5;
             end;
       thus ex x st for y holds Blob[x, y] by 3;
    end;

           == Interpretation I3 is an even easier way to make it true.

T3: (for y ex x st Blob[x, y]) implies (ex x st for y holds Blob[x, y])
    proof
       assume A1: for y ex x st Blob[x, y];
```

```
         consider t;
         A2: for y holds Blob[t, y] proof
             let y be THING;
             thus Blob[t, y] by I3;
         end;
   thus ex x st for y holds Blob[x, y] by A2;
end;
```

9.1 Premises

A formal proof of a formula by itself is just a string of characters. It has no intrinsic meaning. To give the proof meaning we must use the axioms in the environment to provide an interpretation for the predicates and constants in the proof. Arriving at suitable premises for an argument can be very difficult, because it requires one to thoroughly understand the problem at hand. But choosing your premises wisely is most important, for a correct proof means nothing if it is based on inappropriate premises.

(ex49.mse) Consider the following example:

```
environ

    == We interpret the predicates Div and Mult as follows:
    ==           Div[x,y,z] iff x / y = z
    ==           Mult[x,y,z] iff  x * y = z

reserve x,y,z,t,p for Real;

A1: for x,y,z holds Div[x, y, z] iff Mult[z, y, x];

A2: for x,y,z,t holds Div[z, t, x] & Div[z, t, y] implies x=y;

A3: ex p st for x holds Mult[x, p, p];

begin

absurd: for x, y holds x=y proof
    let x, y be Real;
           consider p such that
       B1: for t holds Mult[t, p, p] by A3;
        1: Mult[x, p, p] & Mult[y, p, p] by B1;
        2: Div[p, p, x] & Div[p, p, y] by 1, A1;
        3: x=y by 2, A2;
    thus thesis by 3;
end;
```

Although the proof is perfectly correct, it has a conclusion that is blatantly silly

```
for x,y being Real holds x=y
```

which says that there is only one real number!

The problem with this proof is that one of the premises is inappropriate, and does not accurately describe a property of the real numbers. Specifically, axiom `A1` does not prohibit division by 0. As a result we do not have a faithful interpretation of the real numbers.

Let us consider another case where the above glitch has been fixed but another has been introduced.

(ex50.mse)

```
environ

    == We interpret the predicates Div and Mult as follows:
    ==           Div[x,y,z] iff x / y = z
    ==           Mult[x,y,z] iff  x * y = z

reserve x,y,z,t,p for Real;

    given 0 being Real;

A1: for x,y,z st y <> 0 holds Div[x, y, z] iff Mult[z, y, x];

A2: for x,y,z,t holds Div[z, t, x] & Div[z, t, y] iff x=y;

A3: ex p st for x holds Mult[x, p, p];

    given 1 being Real;

A4: 1<>0;

A5: for x holds Mult[x, 1, x];

begin

absurd: for x holds Mult[0,1,x] proof
    let x be Real;
       a1: 0 = 0;
       a2: Div[x,1,0] by a1, A2;
    thus thesis by A1, a2, A4;
end;
```

Here is still another example with a similar problem. What is the problem?

(ex51.mse)

```
environ
        == We interpret the predicates Neg and Square as follows:
        ==         Neg[x,y] iff y = -x
        ==         Square[x,z] iff  x * x = z

reserve x,y,z,t,p for Real;

A1: for x ex z st Neg[x,z];

A2: for x ex z st Square[x,z];

A3: for x, y, z st Neg[x,y] holds Square[x,z] iff Square[y,z];

A4: for x, y, z st Square[x,z] & Square[y,z] holds x = y;

begin

absurd: for x holds Neg[x,x] proof
    let x be Real;
           consider p such that
      B1: Neg[x,p] by A1;
           consider t such that
      B2: Square[x,t] by A2;
       1: Square[x,t] iff Square[p,t] by B1, A3;
       2: Square[x,t] & Square[p,t] by B2, 1;
       3: x=p by 2, A4;
    thus thesis by B1, 3;
end;
```

Part III
Basics: Sets, Relations, Functions, Induction

10 Simple Set Theory

What is a set? It is simply an unordered collection of things taken from some universe. The notion of set is based on our intuitions about collections of objects, and there are many possible formalizations of this notion.

We will give just one. In our formalization, the universe consists of exactly one sort, called `THING`, of objects. Every object of sort `THING` is a set, and the members or elements of a set are also of sort `THING`. Thus an object can be a viewed as a set or an element of a set depending on your situation.

10.1 Membership

We use one basic predicate `In[THING, THING]`. We interpret `In[x, Y]` as being true if and only if `x` is an element of set `Y`. When not using Mizar, we often use the more common notation $\mathtt{x} \in \mathtt{Y}$.

Example: Suppose `X` is the set $\{1, 2, 3, 4\}$. Then `In[1, X]` is true, and `In[5, X]` is false. Also `In[X, 1]` is false, as is `In[X, X]`.

Sets can contain sets. For example the set $\{\{1\}, \{2, 3\}, \{3, 5\}\}$ contains 3 elements, each of which is a set.

The size of a set X is called its *cardinality*, which we denote in many ways: $|X|$, $\mathrm{card}(X)$, or $\#(X)$. We say a set is finite if its cardinality is a natural number.

10.2 Defining sets

There are many ways to define a set. Some of the more important means are:

1. Enumeration — just list the elements of the set you are defining, as we did above.

2. Specification — give a specific property for elements of another set and choose these that have the property. For example, to specify the set of all persons who are knights we could write

   `Knights` = {`p` | `p` is a `Person` and `Knight[p]` holds}

   This can be read as saying that the set `Knights` contains all things all those members of the set `Person` which satisfy the `Knight` property. In Mizar we could write this as follows

   ```
   given Person, Knights being THING;
   for p being THING holds
        In[p, Knights] iff In[p, Person] & Knight[p]
   ```

   Note how we use a predicate `Knight` to further select those things from the set `Person` that are knights.

3. Construction — use set operations to construct new sets from other sets.

4. Induction — see section 12.

10.3 Equality

Two sets are equal if they contain exactly the same elements. This is expressed formally by the axiom:

```
EqualDef: for A, B being THING holds A=B iff
(for x being THING holds In[x, A] iff In[x, B]).
```

10.4 The Empty Set

Here is the definition of an important set, called `Empty`, which contains no elements.

```
EmptyDef: given Empty being THING;
for x being THING holds not In[x, Empty];
```

That is, there are no elements in the empty set. We often write Ø or {} for the empty set, but we lack this notation in Mizar.

Here is a slightly more complicated set: {Ø} or {{}} which we can define by

```
given SetOfEmpty being THING;
SetOfEmptyDef: for x being THING holds In[x, SetOfEmpty] iff x=Empty;
```

10.5 Subsets

Consider the set $B = \{1, 2, 3, 4, 5\}$ and the set $A = \{1, 2, 3\}$. A has a special relationship to B — all of A's elements are also elements of B. We say that A is a *subset* of B, and write $A \subseteq B$.

The Mizar definition of the subset predicate can be expressed formally by

```
SubsetDef: for A, B being THING holds Subset[A, B] iff
(for x being THING holds In[x, A] implies In[x, B]);
```

That is, $A \subseteq B$ if and only if for all x, $x \in A$ implies $x \in B$.

10.6 Power Set

If you have a set A, you can make many different subsets. The set of all such subsets is called the *power set* of A. We write this as $\mathcal{P}(A)$. Formally we can give the definition

```
PowerSetDef: for A, B being THING holds PowerSet[A, B] iff
(for x being THING holds Subset[x, A] iff In[x, B]);
```

This is saying that B is the power set of A iff every *subset* of A is an *element* of B. For example

$$\mathcal{P}(\{1, 2, 3\}) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$$

For a finite set X we have that $\mathrm{card}(\mathcal{P}(X)) = 2^{\mathrm{card}(X)}$. This will be proven in 12.2.6.

10.7 Union and Intersection

Two important operations on sets are *union* and *intersection.* The union of A and B, written $A \cup B$, is the set consisting of elements that come from A, B, or both.

```
UnionDef: for A, B, C being THING holds Union[A, B, C] iff
(for x being THING holds In[x, A] or In[x, B] iff In[x, C]);
```

Note how the logical operation `or` is used to define union.

The intersection $A \cap B$ of sets A and B consists of all those elements common to both A and B. Its definition looks exactly like union except `or` is changed to `&`.

```
InterDef: for A, B, C being THING holds Inter[A, B, C] iff
(for x being THING holds In[x, A] & In[x, B] iff In[x, C]);
```

10.8 Difference and Complement

The difference between a set A and a set B is the set of all elements of A that are not in B.

```
DiffDef: for A, B, C being THING holds Diff[A, B, C] iff
(for x being THING holds In[x, A] & not In[x,B] iff In[x, C]);
```

Notationally we write $A - B$ or $A \setminus B$.

Note that according to this definition B could contain elements not in A. A more restricted version of difference is called the complement, or relative complement.

```
CompDef: for A, B, C being THING holds
Comp[A, B, C] iff Subset[A, C] & Subset[B, C] & Diff[C, B, A];
```

That is, B is the complement of A with respect to C.

When the universe of discourse is fixed, the complement of a set A with respect to the universe is often denoted A^c or $\bar{A}$.

10.9 Cartesian Products

The Cartesian product of set A with set B, denoted $A \times B$ is the set of all ordered pairs $\langle a, b\rangle$ where a is a member of A and b is a member of B.

For example if $A = \{i, j, k\}$ and $B = \{0, 1\}$ then

$$\begin{aligned} A \times B &= \{\langle i,0\rangle, \langle i,1\rangle, \langle j,0\rangle, \langle j,1\rangle, \langle k,0\rangle, \langle k,1\rangle\} \\ B \times A &= \{\langle 0,i\rangle, \langle 1,i\rangle, \langle 0,j\rangle, \langle 1,j\rangle, \langle 0,k\rangle, \langle 1,k\rangle\} \\ &= \{\langle 0,i\rangle, \langle 0,j\rangle, \langle 0,k\rangle, \langle 1,i\rangle, \langle 1,j\rangle, \langle 1,k\rangle\} \end{aligned}$$

Using an extended Mizar notation we could define a Cartesian product by

```
for A, B, C being THING holds
CarProd[A, B, C] iff (for a, b being THING holds
In[a, A] & In[b, B] iff In[a, b, C])
```

Cartesian products can be among more than two sets, thus we have the generalized notion of a *k-tuple* $\langle x_1, x_2, \ldots, x_k\rangle$ which contains k objects. These tuples are members of a Cartesian product $A_1 \times A_2 \times \ldots \times A_k$ among k sets. It contains all k-tuples where the i-th element of the k-tuple comes from set A_i.

Order is important in a Cartesian product, as is parenthesization. There is a difference between $A \times B \times C$, $A \times C \times B$, and $A \times (B \times C)$. For example:

$$\begin{aligned} &\{0,1\} \times \{a,b\} \times \{1,b\} \\ &= \{\langle 0,a,1\rangle, \langle 0,a,b\rangle, \langle 0,b,1\rangle, \langle 0,b,b\rangle, \langle 1,a,1\rangle, \langle 1,a,b\rangle, \langle 1,b,1\rangle, \langle 1,b,b\rangle\} \end{aligned}$$

while

$$\begin{aligned} &\{0,1\} \times (\{a,b\} \times \{1,b\}) \\ &= \{\langle 0,\langle a,1\rangle\rangle, \langle 0,\langle a,b\rangle\rangle, \langle 0,\langle b,1\rangle\rangle, \langle 0,\langle b,b\rangle\rangle, \langle 1,\langle a,1\rangle\rangle, \langle 1,\langle a,b\rangle\rangle, \langle 1,\langle b,1\rangle\rangle, \langle 1,\langle b,b\rangle\rangle\} \end{aligned}$$

The *projection* operator $|$ is used to break k-tuples apart. In general, if you have a k-tuple $t = \langle x_1, x_2, \ldots, x_k \rangle$ then $t|i = x_i$. That is $t|i$ projects out the i-th component of the k-tuple t.

Of course, we must have $1 \leq i \leq k$. For example $\langle \langle a, b \rangle | 2, \langle a, b \rangle | 1 \rangle = \langle b, a \rangle$. Often we want to use zero-origin indexing, in which case $t|0$ would be the first component of the tuple. (That is, we often want to alter our interpretation of the notation $T|i$. This causes no end of problems.)

10.10 A Puzzle

Here is an interesting set:

```
given U being THING;
for x being THING holds In[x, U];
```

This is the set of everything. Does this set contain itself? Does this even make sense?

Consider the set R which is the set of all sets in U that do not contain themselves as members. We can define R as follows:

```
given R being THING;
for x being THING holds In[x, R] iff In[x, U] & not In[x, x];
```

This is a perfectly reasonable definition.

Now by definition of U, R is a member of U. Furthermore, $R \in R$ or $R \notin R$. But by definition of R, if $R \in R$ then $R \notin R$, and if $R \notin R$ then $R \in R$. Either way we have a contradiction, which means that such a set as U cannot exist.

10.11 DeMorgan's Laws for Sets

Let us fix a set U and consider two subsets A and B of U, along with their complements with respect to U.

If one thinks of $\cap$ as `&`, $\cup$ as `or`, and complement as `not` then DeMorgan's laws can be translated into versions that apply to sets. For example, from the logical relationship

```
not ( A[] & B[] ) iff ( not A[] or not B[] )
```

comes the set relationship

$$(A \cap B)^c = A^c \cup B^c$$

which expressed using set difference is

$$(U - (A \cap B)) = (U - A) \cup (U - B)$$

How would you prove such a law in Mizar? It is tedious even to state the relationship between the sets. We must somehow represent the formula involving set operations as a logical formula. One way to do it is to introduce a variable for each piece of the formula as follows:

```
for A, B, U being THING holds ex L, R, S1, T1, T2 being THING st
Inter[A, B, S1] & Diff[U, S1, L] &
Diff[U, A, T1] & Diff[U, B, T2] & Union[T1, T2, R] &
L = R
```

All this is necessary just to say that the left hand side equals the right hand side in the set formula above.

One would think that we could now proceed to prove De Morgan's Law. But we can't with just the set theory axioms we have so far, at least not directly.

For example, we know what it means to be a subset or difference from the definitions `SubsetDef` and `DiffDef`. But how do we know that such objects even exist? We need to add some existence axioms in order to permit existential instantiation. For example:

```
for U, A being THING ex B being THING st Diff[U, A, B].
```

Then we will be able to say things like "consider B being the difference of U and A".

The other problem we encounter if we try to prove De Morgan's Law is that at some point we will need to conclude something like

```
Diff[U, A, X]
Diff[U, A, Y]
```

thus

```
X = Y
```

But in general this is not true. It is only true if we know that the difference of a set is unique. So we will need to prove things like

```
for U, A, B, C being THING holds
Diff[U, A, B] & Diff[U, A, C] implies B = C.
```

10.12 Mizar Axioms and Examples for Simple Set Theory

(sets.mse)

```
environ == axioms for simple set theory

== set membership
== In[THING, THING]  - where In[x, A] means x is an element of set A

== equality
== A=B - two sets are equal if they have the same elements

EqualDef: for A, B being THING holds A=B iff
    (for x being THING holds In[x, A] iff In[x, B]);


== the empty set and its definition

given Empty being THING;
EmptyDef: for x being THING holds not In[x, Empty];


== subsets
== Subset[THING, THING] - where Subset[A, B] means A is a subset of B

SubsetDef: for A, B being THING holds Subset[A, B] iff
    (for x being THING holds In[x, A] implies In[x, B]);


== PowerSet[THING, THING] - where PowerSet[A, B] means B is the power set
== of A

PowerSetDef: for A, B being THING holds PowerSet[A, B] iff
    (for x being THING holds Subset[x, A] iff In[x, B]);


== set operations

== union
== Union[THING, THING, THING] - where Union[A, B, C] means C is the
== union of A and B

UnionDef: for A, B, C being THING holds Union[A, B, C] iff
    (for x being THING holds In[x, A] or In[x, B] iff In[x, C]);

== intersection
== Inter[THING, THING, THING] - where Inter[A, B, C] means C is the
== intersection of A and B

InterDef: for A, B, C being THING holds Inter[A, B, C] iff
    (for x being THING holds In[x, A] & In[x, B] iff In[x, C]);
```

```
== difference
== Diff[THING, THING, THING] - where Diff[A, B, C] means C is the set
== difference of A with B

DiffDef: for A, B, C being THING holds Diff[A, B, C] iff
    (for x being THING holds In[x, C] iff In[x, A] & not In[x, B]);


== Symmetric difference
== SymDiff[THING, THING, THING] - where SymDiff[A,B,C] means C is
== the symmetric difference of A and B

SymDiffDef: for A,B,C being THING holds SymDiff[A,B,C] iff
        (for x being THING holds In[x,C] iff
            (In[x,A] or In[x,B]) & not (In[x,A] & In[x,B]));

== complement
== Comp[THING, THING, THING] - where Comp[A, B, U] means B is the
== complement of A with respect to U

CompDef: for A, B, U being THING holds
    Comp[A, B, U] iff Subset[A, U] &  Subset[B, U] & Diff[U, A, B];

reserve A, B, C, D, E, U, V, W for THING;
reserve x, y, z for THING;

begin

== Show that Empty is a subset of every set.

for A being THING holds Subset[Empty, A] proof
    let A be THING;
            == we need to use the definition of subset
    1: Subset[Empty, A] iff
         (for x being THING holds In[x, Empty] implies In[x, A]) by SubsetDef;
            == now work with the individual elements of Empty
    2: for x being THING holds In[x, Empty] implies In[x, A] proof
        let x be THING;
        assume A1: In[x, Empty];
                3: not In[x, Empty] by EmptyDef;
            == A1 and 3 contradict each other, so now everything follows
        thus thesis by A1, 3;
       end;
    thus Subset[Empty, A] by 1, 2;
end;


== Show that A is a subset of itself (a similar proof to the previous one).

for A being THING holds Subset[A, A] proof
```

```
    let A be THING;
            == we need to use the definition of subset
    1: Subset[A, A] iff
             (for x being THING holds In[x, A] implies In[x, A]) by SubsetDef;
            == now work with the individual elements of A
    2: for x being THING holds In[x, A] implies In[x, A] proof
        let x be THING;
        thus In[x, A] implies In[x, A];
       end;
    thus Subset[A, A] by 1, 2;
end;


== Show that if x is a member of A, and A is a subset of B then
== x is a member of B.

for x, A, B being THING holds In[x, A] & Subset[A, B] implies In[x, B] proof
    let x, A, B be THING;
    assume A1: In[x, A] & Subset[A, B];
    thus In[x, B] by A1, SubsetDef;
end;


== Show that when A and B are both subsets of U, then difference behaves
== like subtraction.

for A, B, U being THING holds Subset[A, U] & Subset[B, U]
                               implies (Diff[U, A, B] iff Diff[U, B, A]) proof
    let A, B, U be THING;
    assume A1: Subset[A, U] & Subset[B, U];
    Tlr: Diff[U, A, B] implies Diff[U, B, A] proof
          assume A2: Diff[U, A, B];
          1: for x being THING holds In[x, B] iff In[x, U] & not In[x, A]
                                                                by A2, DiffDef;
          2: for x being THING holds In[x, A] iff In[x, U] & not In[x, B] proof
              let x be THING;
               3: In[x, B] iff In[x, U] & not In[x, A]  by 1;
               4: In[x, B] implies In[x, U] by A1, SubsetDef;
               5: In[x, A] implies In[x, U] by A1, SubsetDef;
              thus thesis by 3, 4, 5;
             end;
          thus Diff[U, B, A] by 2, DiffDef;
         end;

    Trl: Diff[U, B, A] implies Diff[U, A, B] proof
          assume A2: Diff[U, B, A];
          1: for x being THING holds In[x, A] iff In[x, U] & not In[x, B]
                                                                by A2, DiffDef;
          2: for x being THING holds In[x, B] iff In[x, U] & not In[x, A] proof
              let x be THING;
               3: In[x, A] iff In[x, U] & not In[x, B]  by 1;
               4: In[x, A] implies In[x, U] by A1, SubsetDef;
               5: In[x, B] implies In[x, U] by A1, SubsetDef;
              thus thesis by 3, 4, 5;
            end;
          thus Diff[U, A, B] by 2, DiffDef;
         end;

    thus thesis by Tlr, Trl;
end;

Exercise1: == Intersection of two sets is contained in their union.

for A, B, C, D st Inter[A,B,C] & Union[A,B,D] holds Subset[C,D] proof
    let A,B,C,D be THING;
    thus Inter[A,B,C] & Union[A,B,D] implies Subset[C,D] proof
        assume 1: Inter[A,B,C] & Union[A,B,D];
               2: for x being THING holds In[x,C] implies In[x,D] proof
                   let y be THING;
                   thus In[y,C] implies In[y,D] proof
                       assume 3: In[y,C];
                              4: In[y,A] & In[y,B] by 3,1,InterDef;
                              5: In[y,A] or In[y,B] by 4;
                       thus thesis by 5,1,UnionDef;
                   end;
               end;
        thus Subset[C,D] by SubsetDef,2;
    end;
end;

Exercise2: == What is the intersection of the empty set with a set?

for A holds Inter[A, Empty, Empty] proof
    let A be THING;
    0: for x being THING holds In[x,A] & In[x,Empty] iff In[x,Empty] proof
           let y be THING;
           thus In[y,A] & In[y,Empty] implies In[y,Empty];
           thus In[y,Empty] implies In[y,A] & In[y,Empty] proof
               1: not In[y,Empty] proof
                     assume 2: In[y,Empty];
                            3: y<>y by EmptyDef,2;
                     thus contradiction by 3;
                 end;
               thus thesis by 1;
           end;
       end;
    thus thesis by InterDef,0;
end;

Exercise3: == What is the union of the empty set with a set?

for A holds Union[A, Empty, A] proof
```

```
    let A be THING;
    0: for x being THING holds In[x,A] or In[x,Empty] iff In[x,A] proof
           let y be THING;
           thus In[y,A] or In[y,Empty] iff In[y,A] proof
               1: In[y,A] or In[y,Empty] implies In[y,A] proof
                   assume 2: In[y,A] or In[y,Empty];
                   thus In[y,A] proof
                       assume 3: not In[y,A];
                              4: In[y,Empty] by 2,3;
                              5: y<>y by 4, EmptyDef;
                       thus contradiction by 5;
                   end;
               end;
               2: In[y,A] implies In[y,A] or In[y,Empty];
               thus thesis by 1,2;
           end;
       end;
    thus thesis by UnionDef,0;
end;

Exercise4: == The subset relation is transitive.

for A, B, C st Subset[A,B] & Subset[B,C] holds Subset[A,C] proof
    let A,B,C be THING;
    thus Subset[A,B] & Subset[B,C] implies Subset[A,C] proof
        assume 1: Subset[A,B] & Subset[B,C];
               2: for x being THING holds In[x,A] implies In[x,C] proof
                      let y be THING;
                      thus In[y,A] implies In[y,C] proof
                         assume 3: In[y,A];
                                4: In[y,B] by 1,3,SubsetDef;
                         thus In[y,C] by 1,4,SubsetDef;
                      end;
                  end;
        thus thesis by 2,SubsetDef;
    end;
end;

Exercise5: == How are the set difference and symmetric difference related?

for A, B, C, D, E st Diff[A,B,C] & Diff[B,A,D] & SymDiff[A,B,E]
            holds Union[C,D,E] proof

let A,B,C,D,E be THING;
assume 1: Diff[A,B,C] & Diff[B,A,D] & SymDiff[A,B,E];
       2: for x being THING holds In[x,C] or In[x,D] iff In[x,E] proof
              let y be THING;
              thus In[y,C] or In[y,D] iff In[y,E] proof
                  thus
                  In[y,C] or In[y,D] implies In[y,E] proof
                      assume 3: In[y,C] or In[y,D];
```

```
                              4: In[y,A] & not In[y,B] or In[y,B] & not In[y,A]
                                  by 1,3,DiffDef;
                              5: (In[y,A] or In[y,B]) & not (In[y,A] & In[y,B])
                                 by 4;
                        thus thesis by 1,SymDiffDef,5;
                    end;
                    thus
                    In[y,E] implies In[y,C] or In[y,D] proof
                        assume 6: In[y,E];
                               7: (In[y,A] or In[y,B]) & not (In[y,A] & In[y,B])
                                  by 1,6,SymDiffDef;
                               8: In[y,A] & not In[y,B] or In[y,B] & not In[y,A]
                                  by 7;
                        thus thesis by 1,DiffDef,8;
                    end;
                end;
            end;
            thus Union[C,D,E] by UnionDef,2;
end;

Exercise6: == Union is commutative.

for A, B, C, D st Union[A,B,C] & Union[B,A,D] holds C = D proof
    let A,B,C,D be THING;
    thus Union[A,B,C] & Union[B,A,D] implies C=D proof
        assume 1: Union[A,B,C] & Union[B,A,D];
               2: for x being THING holds In[x,C] iff In[x,D] proof
                      let y be THING;
                      thus In[y,C] implies In[y,D] proof
                          assume 3: In[y,C];
                                 4: In[y,A] or In[y,B] by 1,3,UnionDef;
                          thus In[y,D] by 1,4,UnionDef;
                      end;
                      thus In[y,D] implies In[y,C] proof
                          assume 5: In[y,D];
                                 6: In[y,B] or In[y,A] by 1,5,UnionDef;
                          thus In[y,C] by 1,6,UnionDef;
                      end;
                  end;
        thus thesis by 2,EqualDef;
    end;
end;

Exercise7: == The set difference of two sets is unique.

for A, B, C, D st Diff[A,B,C] & Diff[A,B,D] holds C = D proof
    let A,B,C,D be THING;
    thus Diff[A,B,C] & Diff[A,B,D] implies C=D proof
        assume 1: Diff[A,B,C] & Diff[A,B,D];
               2: for x being THING holds In[x,C] iff In[x,D] proof
                      let y be THING;
                      thus In[y,C] iff In[y,D] proof
                          thus In[y,C] implies In[y,D] proof
                              assume 3: In[y,C];
                                     4: In[y,A] & not In[y,B] by 1,3,DiffDef;
                              thus In[y,D] by 1,4,DiffDef;
                          end;
                          thus In[y,D] implies In[y,C] proof
                              assume 5: In[y,D];
                                     6: In[y,A] & not In[y,B] by 1,5,DiffDef;
                              thus In[y,C] by 1,6,DiffDef;
                          end;
                      end;
                  end;
        thus thesis by 2,EqualDef;
    end;
end;

Exercise8: == A relationship between intersection and union.

for A, B, C, D st Inter[A,B,C] & Union[A,B,D] & C = D holds A = B proof
    let A,B,C,D be THING;
    thus Inter[A,B,C] & Union[A,B,D] & C = D implies A = B proof
        assume 1: Inter[A,B,C] & Union[A,B,D] & C = D;
               2: for x being THING holds In[x,A] iff In[x,B] proof
                      let y be THING;
                      thus In[y,A] iff In[y,B] proof
                          thus In[y,A] implies In[y,B] proof
                              assume 3: In[y,A];
                                     4: In[y,C] by 1,3,UnionDef;
                              thus In[y,B] by 1,4,InterDef;
                          end;
                          thus In[y,B] implies In[y,A] proof
                              assume 5: In[y,B];
                                     6: In[y,C] by 1,5,UnionDef;
                              thus In[y,A] by 1,6,InterDef;
                          end;
                      end;
                  end;
        thus A=B by 2,EqualDef;
    end;
end;

Exercise9: == If a set has at least two elements then for every element
           == in the set there exists an element in the set different
           == than the first one.

for A st (ex x, y st x<>y & In[x,A] & In[y,A])
  holds for z st In[z,A] holds ex u being THING st In[u,A] & u<>z  proof
let A be THING;
thus (ex x, y st x<>y & In[x,A] & In[y,A]) implies
       (for z st In[z,A] holds ex u being THING st In[u,A] & u<>z)  proof
    assume 1: ex x, y st x<>y & In[x,A] & In[y,A];
    thus for z st In[z,A] holds ex u being THING st In[u,A] & u<>z  proof
        let z be THING;
        thus In[z,A] implies (ex u being THING st In[u,A] & u<>z)   proof
            assume 2: In[z,A];
                   consider x,y being THING such that
                   3: x<>y & In[x,A] & In[y,A] by 1;
                   4: x=z implies thesis proof
                       assume 41: x=z;
                              42: In[y,A] & y<>z by 3,41;
                       thus ex u being THING st In[u,A] & u<>z by 42;
                   end;
                   5: x<>z implies thesis proof
                       assume 51: x<>z;
                              52: In[x,A] & x<>z by 3,51;
                       thus ex u being THING st In[u,A] & u<>z by 52;
                   end;
            thus thesis by 4,5;
        end;
    end;
end;
end;

Exercise10: == For any two sets, if there is an element that is in one
            == set but not in the other, then their symmetric difference
            == is not empty.

for A, B st ex x st In[x,A] & not In[x,B] holds
                                      for C st SymDiff[A,B,C] holds C <> Empty
 proof
  let A,B be THING;
  thus (ex x st In[x,A] & not In[x,B]) implies
                                     (for C st SymDiff[A,B,C] holds C <> Empty)
  proof
   assume 1: ex x st In[x,A] & not In[x,B];
   thus for C st SymDiff[A,B,C] holds C <> Empty
   proof
    let C be THING;
    thus SymDiff[A,B,C] implies C <> Empty
    proof
     assume 2: SymDiff[A,B,C];
     thus C <> Empty proof
           assume 3: C=Empty;
                     consider y being THING such that
                  4: In[y,A] & not In[y,B] by 1;
                  5: not In[y,C] by 3,EmptyDef;
                  6: not((In[y,A] or In[y,B]) & not(In[y,A] & In[y,B]))
                                                          by SymDiffDef,2,5;
                  7: not (In[y,A] or In[y,B]) or (In[y,A] & In[y,B]) by 6;
                  8: In[y,A] or In[y,B] by 4;
                  9: In[y,A] & In[y,B] by 7,8;
```

```
          thus contradiction by 4,9;
         end;
   end;
  end;
 end;
end;
```

11 Functions and Relations

11.1 Functions

One of the key notions of computing is that of taking an input and transforming it into some sort of output. Often the transformation has the important property that it is *functional* — if the transformation is applied to a particular input and it results in some output, then doing the transformation again at a later time on the same particular input results in the same output.

A *function* is a mapping from one set of objects called the *domain type*, to another set of objects called the *range type*. The specification of the mapping is via a set of ordered pairs called the *graph* of the function.

We formally specify a function f as a 3-tuple $f = \langle D_f, R_f, G_f \rangle$ where D_f is the set that gives the domain type of f, R_f is the set that gives the range type of f, and G_f is a subset of $D_f \times R_f$ that specifies the graph of f. The set G_f consists of ordered pairs of the form $\langle x, y \rangle$ with the special *functional property* that no two ordered pairs have the same x value.

To define a function we must provide all three components: the domain type, the range type, and the graph. One way of doing this is by explicitly listing all three sets, as in the following definition of the logical not function:

$$\mathit{Not} = \langle \mathit{Boolean}, \mathit{Boolean}, \{\langle \mathit{true}, \mathit{false}\rangle, \langle \mathit{false}, \mathit{true}\rangle\}\rangle$$

In a simple case like this it is often easier to supply the graph in a tabular form:

G_{Not} :

x	$\mathit{Not}(x)$
true	false
false	true

Here is the formal definition of the function that returns the sign of a real number:

$$\mathit{Sgn} = \langle \mathit{real}, \mathit{integer}, G_{\mathit{Sgn}} \rangle$$

where

$$G_{\mathit{Sgn}} = \{\langle x, -1\rangle \mid x < 0\} \cup \{\langle 0, 0\rangle\} \cup \{\langle x, 1\rangle \mid x > 0\}$$

Here is a definition of division in terms of multiplication:

$$\mathit{Divide} = \langle \mathit{real} \times \mathit{real}, \mathit{real}, G_{\mathit{Divide}} \rangle$$

where

$$G_{\mathit{Divide}} = \{\langle\langle x, y\rangle, z\rangle \mid x = y * z\}$$

The preceding definitions simply stated the input/output relationship of the function, they gave no explicit instructions for how to compute the function. On the other hand, the following functions are given by providing a rule, in this case a Pascal program, for computing a member of the graph of the function given the first component. We have to deduce the formal definition of the functions from our knowledge of Pascal.

```
function $Square(x : integer) : integer$;
begin
    $Square := x * x$;
end;

function $Mult(x : integer; y : integer) : integer$;
begin
    $Mult := x * y$;
end;

function $Inverse(x : real) : real$;
begin
    $Inverse := 1.0/x$;
end;

function $Cube(x : real) : real$;
begin
    $Cube := x * x * x$;
end;
```

Note how the Pascal type declaration provides the definition of the domain type and range type of each of the functions.

Function	Domain Type	Range type
Not	*Boolean*	*Boolean*
Sgn	*real*	*integer*
Divide	$real \times real$	*real*
Square	*integer*	*integer*
Mult	$integer \times integer$	*integer*
Inverse	*real*	*real*
Cube	*real*	*real*

A function is intended to be applied only to objects from its domain type, and is guaranteed to return only objects of its range type. Specifying the domain and range type of a function enables one to do *static type checking*.

As one knows from everyday programming, just because a function is applied to an argument of the correct type, it does not mean that the function will actually return a value. For example, $Inverse(0.0)$ will generate a divide by 0 error. The subset of the domain type on which the function is defined is called its *domain*.

Similarly, just because a function can return a value of some type, it does not mean that it will do so for each value in the type. That is, each element of the range type may not necessarily be associated with an element of the domain that transforms to it. For example, no value x of the domain type of *Square* can be found such that $Square(x) = 2$. The subset of the range type which can actually be produced by a function is called its *range*.

The *domain* of f, written domain(f), is a subset of the domain type D_f and given by:

$$\text{domain}(f) = \{x \in D_f \mid \exists\, y : y \in R_f : \langle x, y\rangle \in G_f\}$$

The *range* of f, written range(f), is a subset of the range type R_f and given by

$$\text{range}(f) = \{y \in R_f \mid \exists\, x : x \in D_f : \langle x, y\rangle \in G_f\}$$

A function *maps* elements in the domain to elements in the range. If $\langle x, y\rangle \in G_f$ we say that x is mapped to y, and we write $f(x) = y$. Remember, for every $x \in$ domain(f) there exists exactly one y such that $\langle x, y\rangle \in G_f$.

A function is itself an object, and it could be the argument to a function. For this reason we often find it useful to specify the type of the function f using the notation $f : D_f \mapsto R_f$. This can be read as "f is a map from D_f to R_f".

All functions are *partial*. That is, they are not necessarily defined on all members of their domain type. If $f(x)$ is defined on every element in the domain type then we say that function f is *total*. That is, domain(f) = D_f. Note that partial is a superset of total, and does not mean not-total. The functions *Square*, *Mult*, and *Cube* are total, while *Inverse* is not total.

A function is *one-to-one* (or 1–1, or an *injection*) iff no two elements of domain(f) are mapped to the same element of range(f). That is, f is 1–1 iff for any $x_1, x_2 \in$ domain(f), whenever $f(x_1) = f(x_2)$ then $x_1 = x_2$. Generally functions are *many-to-one*. Note that many-to-one is a superset of 1–1, and does not mean not-1–1. The functions *Inverse* and *Cube* are 1-1, while the functions *Square* and *Mult* are not 1-1.

A function is *onto* (or a *surjection*) iff range(f) = R_f. That is, if every element of the range type is mapped to by some element of domain(f). The functions *Mult* and *Cube* are onto, while *Square* and *Inverse* are not.

A function f is *invertible* iff it is total, 1–1, and onto. An invertible function is also called a *bijection* or a 1–1 correspondence. The function *Cube* is invertible. None of *Square*, *Mult*, or *Inverse* are invertible.

The properties of the functions we have just introduced are summarized below:

Function	Total	Onto (surjection)	1-1 (injection)	Invertible (bijection)
Not	Y	Y	Y	Y
Sgn	Y	N	N	N
Divide	N	Y	N	N
Square	Y	N	N	N
Mult	Y	Y	N	N
Inverse	N	N	Y	N
Cube	Y	Y	Y	Y

As an exercise, determine the domain type, range type, domain, and range of the following function:

```
function $h(\langle i, j\rangle) : Int \times Int \mapsto Int \times Int$
    $u, v : Int$;
    $u \leftarrow i$; $v \leftarrow j$;
    while $(u > 0)$ do
        $v \leftarrow v + u$;
        $u \leftarrow u - 1$;
    end;
    return $\langle u, v\rangle$;
end;
```

11.2 Equality of Functions

The formal definition of a function gives us a precise way of stating when two functions are equal — when they have the same formal definition. That is, two functions are equal when they have the same domain type, same range type, and identical graphs. For example if $f = \langle D_f, R_f, G_f\rangle$ and $g = \langle D_g, R_g, G_g\rangle$ then $f = g$ exactly when $D_f = D_g$, $R_f = R_g$ and $G_f = G_g$.

Often we talk about functions with the same domain and range types, for example we may have $f : X \mapsto Y$ and $g : X \mapsto Y$. In this situation we have that $f = g$ when they are equal pointwise. That is,

$$f = g \text{ iff } (\forall\, x : x \in X : f(x) = g(x))$$

What does the above mean if f and g are not total functions?

11.3 Function Composition

Often one wishes to do a sequence of transformations where the output of one transformation is input to the next. For example, one might attempt to compute *Inverse*(*Mult*(x, y)) for arbitrary x and y.

A similar example is a sequence of statements in a program:

```
$t := x * y$;
$u := 1.0/t$;
```

where x, y are integers and t, u are reals.

Frequently, we want to talk about the effect of the sequence of transformations independently from the context that they occur. That is, without knowing what values x and y have. The net effect of the sequence of transformations can itself be expressed as a function.

This function results from composing each of the individual functions. Composition is an operation that takes two functions and produces a new one.

In direct analogy to the use of ";" in Pascal, we express the new function created by composing f and g by the notation $(f; g)$, which we can read as "f followed by g".

The composition of two functions is not always possible. For the composition f followed by g to make sense, the range type of f must be "compatible" with the domain type of

g in that the values returned by f must be acceptable inputs to the function g. This compatibility can take many forms depending on our goals.

One form of compatibility occurs when we have only simple static type checking. Then we usually require that

$$\text{range type of } f = \text{ domain type of } g$$

A less restrictive form can be used when we have *subtypes.* Then we can relax the static type conditions to

$$\text{range type of } f \subseteq \text{ domain type of } g$$

For example, integers can be a subtype of the reals so we can compose *Mult* followed by *Inverse* to obtain (*Mult*; *Inverse*).

If we want to ensure that the function $(f;g)$ has the same domain as f, then we need to have

$$\text{range}(f) \subseteq \text{domain}(g)$$

In general, the composition $(f;g)$ has the following properties:

$$\text{domain type of } (f;g) = \text{ domain type of } f$$

and

$$\text{range type of } (f;g) = \text{ range type of } g$$

The domain of $(f;g)$ is given by

$$\text{domain}(f;g) = \{x \in \text{domain}(f) \mid f(x) \in \text{domain}(g)\}$$

The range of $(f;g)$ is given by

$$\text{range}(f;g) = \{y \in \text{range}(g) \mid \exists\, x : x \in \text{domain}(f;g) \;\&\; y = g(f(x))\}$$

The composition operation is defined exactly by:

$$\forall\, x : x \in \text{domain}((f;g)) : (f;g)(x) = g(f(x))$$

11.4 Relations

Every predicate can be thought of as a function from the Cartesian product of the sorts of its arguments to the set *Boolean*. For example, suppose there is a predicate `Tells`, where `Tells[p,s]` means that `p`, of sort `Person`, says `s`, of sort `Statement`. Then `Tells` can be thought of as a function from `Person` $\times$ `Statement` to $\{T, F\}$. As a function `Tells[p,s]` returns true when the predicate holds, and false when it does not.

Now assuming that the predicate is always defined, the predicate can return only two values, and so it can be described in a very simple way — just identify the subset of possible arguments that cause the predicate to return true. Any inputs not in this subset must cause the predicate to return false.

For example, in the case of the `Tells` predicate, the possible inputs come from the set $D =$ `Person` $\times$ `Statement`. We could define the set G, with $G \subseteq D$, of ordered pairs for which the predicate holds, say

$$G = \{\langle Fred, Hello\rangle, \langle Bart, OhNo\rangle, \ldots\}$$

The predicate `Tells` is then fully described by the pair of sets $\langle D, G\rangle$.

This approach to describing predicates works whether the predicate has 0, 1, or more arguments. (Note the difference between the set $\{\langle\rangle\}$ containing the empty tuple, and the empty set $\{\}$.) All that is needed to fully describe the predicate is to give the set D of possible inputs, and the subset G of satisfying inputs. Any given pair from G describes values that are related by the fact that they satisfy the predicate. This leads us to the general notion of a *relation*.

A *k-ary relation* on sets $A_1, A_2, \ldots, A_k$ is a subset of $A_1 \times A_2 \times \ldots \times A_k$. We say that the *domain* of the relation is $A_1 \times \ldots \times A_k$. To be precise we specify a k-ary relation R as an ordered pair:

$$R = \langle D, G\rangle$$

where $D = A_1 \times \ldots \times A_k$ and $G \subseteq D$. This notation emphasizes that to fully describe a relation you need to specify both the domain and the subset which gives the related tuples.

For example the set $\{\langle 0, a, 1\rangle, \langle 1, a, b\rangle\}$ is a 3-ary relation on the domain $\{0, 1\} \times \{a, b\} \times \{1, b\}$. The set $\{\langle 0, \langle a, 1\rangle\rangle, \langle 1, \langle a, b\rangle\rangle\}$ is a binary relation on the domain $\{0, 1\} \times (\{a, b\} \times \{1, b\})$. The *input/output* relation of a program is the binary relation that pairs each input to a program with the possible outputs that can be generated by that input.

11.5 Mizar Axioms and Examples for Binary Relations

(binrels.mse)

```
environ

== All binary relations we talk about here are defined on a
== certain, fixed, but otherwise unknown Set of THINGs.

 reserve r, r', r'' for BinRelation;
 reserve x, x', x'', y, y', y'', z, z',z'' for THING;
 given 1, 2, 3, 4, 5, 6, 7, 8 being THING;

== Here is a list of definitions of properties of binary relations.

ReflexivityDef:
  for r holds Reflexive[r] iff (for x holds In[x,x,r]);

IrReflexivityDef:
  for r holds irReflexive[r] iff not (ex x st In[x,x,r]);

SymmetryDef:
  for r holds Symmetric[r] iff
              (for x, y st In[x,y,r] holds In[y,x,r]);

ASymmetryDef:
  for r holds aSymmetric[r] iff
              (for x, y st In[x,y,r] holds not In[y,x,r]);

AntiSymmetryDef:
  for r holds antiSymmetric[r] iff
              (for x, y st In[x,y,r] & In[y,x,r] holds x=y);

TransitivityDef:
  for r holds Transitive[r] iff
              (for x, y, z st In[x,y,r] & In[y,z,r] holds In[x,z,r]);

ConnectednessDef:
  for r holds Connected[r] iff
              (for x, y holds In[x,y,r] or In[y,x,r]);

== Since binary relations are just sets of pairs, let us define some
== set operations on them.
== (Note the change of notation from the previous assignment.)


== r is included in r'
InclusionDef:
  for r,r' holds Includes[r,r'] iff
                 (for x, y st In[x,y,r] holds In[x,y,r']);

RelEqualityDef:
  for r,r' st Includes[r,r'] & Includes[r',r] holds r = r';

== r is the intersection of r' and r''
IntersectionDef:
  for r,r',r'' holds Inter[r,r',r''] iff
           (for x, y holds In[x,y,r'] & In[x,y,r''] iff In[x,y,r]);

== r is the union of r' and r''
UnionDef:
  for r,r',r'' holds Union[r,r',r''] iff
```

```
           (for x, y holds In[x,y,r'] or In[x,y,r''] iff In[x,y,r]);

== And here is an operation very characteristic for binary relations,
== namely, composition.

== r is the composition: r' followed by r''
CompositionDef:
  for r, r', r'' holds Comp[r,r',r''] iff
      (for x, z holds In[x,z,r] iff (ex y st In[x,y,r'] & In[y,z,r'']));

begin

Exercise1:
for r st Connected[r] holds not aSymmetric[r] proof
    let r be BinRelation;
    assume 1: Connected[r];
           2: In[1,1,r] by 1,ConnectednessDef;
    thus not aSymmetric[r] by 2,ASymmetryDef;
end;

Exercise2:
now let r be BinRelation;

        assume a1: In[1,2,r] & In[2,1,r];

        thus irReflexive[r] implies not Transitive[r] proof
            1: Transitive[r] implies not irReflexive[r] proof
                assume 2: Transitive[r];
                       3: In[1,1,r] by 2,TransitivityDef,a1;
                thus not irReflexive[r] by 3,IrReflexivityDef;
            end;
            thus thesis by 1;
        end;
end;

Exercise3:
now let r be BinRelation;

        assume a1: In[1,2,r] & In[3,1,r] &  In[2,3,r];

        thus Transitive[r] implies not aSymmetric[r] proof

           assume 1: Transitive[r];
                  2: In[1,3,r] by TransitivityDef,a1,1;
           thus not aSymmetric[r] by ASymmetryDef,a1,2;

        end;

       thus aSymmetric[r] implies not Transitive[r] proof

           assume 3: aSymmetric[r];
```

```
                 4: not In[1,3,r] by ASymmetryDef,a1,3;
          thus not Transitive[r] by TransitivityDef,a1,4;

       end;
end;

Exercise4:
now let r be BinRelation;

        assume a1: In[2,3,r] &  In[3,4,r] &  In[1,5,r] &
            not In[1,4,r] &  not In[2,5,r];

        thus Connected[r] implies not Transitive[r] proof

            assume 1: Connected[r];
                   2: In[4,1,r] by ConnectednessDef,a1,1;
            thus not Transitive[r] proof
                assume 3: Transitive[r];
                       4: In[4,5,r] by TransitivityDef,2,3,a1;
                       5: In[2,4,r] by 3,TransitivityDef,a1;
                       6: In[2,5,r] by 3,TransitivityDef,4,5;
                thus contradiction by a1,6;
            end;
        end;
end;

Exercise5:
for r,r',r'' st Transitive[r'] & Transitive[r''] & Inter[r,r',r'']
        holds Transitive[r] proof

    let r,r',r'' be BinRelation;
    assume 1: Transitive[r'] & Transitive[r''] & Inter[r,r',r''];

    2: for x, y, z st In[x,y,r] & In[y,z,r] holds In[x,z,r] proof

        let x,y,z be THING;
        assume 3: In[x,y,r] & In[y,z,r];
               4: In[x,y,r'] & In[y,z,r'] by 1,3,IntersectionDef;
               5: In[x,z,r'] by 1,4,TransitivityDef;
               6: In[x,y,r''] & In[y,z,r''] by 1,3,IntersectionDef;
               7: In[x,z,r''] by 1,6,TransitivityDef;
        thus In[x,z,r] by 1,5,7,IntersectionDef;
    end;

    thus Transitive[r] by 1,2,TransitivityDef;
end;

Exercise6:
for r, r', r'' st Symmetric[r'] & Symmetric[r''] & Union[r,r',r'']
        holds Symmetric[r] proof
    let r,r',r'' be BinRelation;
    assume 1: Symmetric[r'] & Symmetric[r''] & Union[r,r',r''];

           2: for x, y st In[x,y,r] holds In[y,x,r] proof

                let x,y be THING;
                assume 3: In[x,y,r];
                       4: In[x,y,r'] or In[x,y,r''] by 1,3,UnionDef;
                       5: In[y,x,r'] or In[y,x,r''] by 1,4,SymmetryDef;
                thus In[y,x,r] by 1,5,UnionDef;

            end;

    thus Symmetric[r] by 2,SymmetryDef;

end;

Exercise7:
now let r, r1, r2, r3, r4 be BinRelation;
    assume a1: In[1,2,r] & In[2,3,r] & In[3,4,r] & In[4,5,r] & In[5,6,r];
    assume c1: Comp[r1,r,r];
    assume c2: Comp[r2,r,r1];
    assume c3: Comp[r3,r,r2];
    assume c4: Comp[r4,r,r3];

    thus In[1,6,r4] proof
        1: In[4,6,r1] by a1,c1,CompositionDef;
        2: In[3,6,r2] by a1,c2,1,CompositionDef;
        3: In[2,6,r3] by a1,c3,2,CompositionDef;
        thus thesis by a1,c4,3,CompositionDef;
    end;
end;

now

== Exercises 8, 9, and 10 concern the notion of
== isomorphism of binary relations.

let A, B be THING;

let R, S be BinRelation;
assume  R1: for x, y st In[x,y,R] holds In[x,A] & In[y,A];
assume  S1: for x, y st In[x,y,S] holds In[x,B] & In[y,B];
                          == i.e. R is defined on A and S is defined on B.

== We assume the existence of a bijective function, named Iso,
== from A to B. We will write Iso[x,y] to say that Iso maps x to y
== (unfortunately Mizar MSE does not provide us with functional notation).

== Iso is a function.
assume F: for x, y, z st In[x,A] & In[y,B] & In[z,B] holds
```

```
                        Iso[x,y] & Iso[x,z] implies y=z;

== Iso is a total function.
assume T: for x st In[x,A] ex x' st In[x',B] & Iso[x,x'];

== Iso is surjective (onto).
assume S: for x' st In[x',B] ex x st In[x,A] & Iso[x,x'];

== Iso is injective (1-1);
assume I: for x, y, z st In[x,A] & In[y,A] & In[z,B] holds
                        Iso[x,z] & Iso[y,z] implies x=y;

== And now we assume that Iso establishes an isomorphism between
== R and S. That is, R and S differ superficially only.

assume ISO: for x,y,x',y' st
   In[x,A] & In[x',B] & In[y,A] & In[y',B] & Iso[x,x'] & Iso[y,y']
   holds In[x,y,R] iff In[x',y',S];

thus
Exercise8:
Symmetric[R] implies Symmetric[S] proof

    assume 1: Symmetric[R];
    thus Symmetric[S] proof
        2: for x, y holds In[x,y,S] implies In[y,x,S] proof

            let x',y' be THING;
            assume 3: In[x',y',S];
                   4: In[x',B] & In[y',B] by 3,S1;

                   5: ex x st In[x,A] & Iso[x,x'] by 4,S;
                   consider x being THING such that
                   6: In[x,A] & Iso[x,x'] by 5,S;

                   7: ex x st In[x,A] & Iso[x,y'] by 4,S;
                   consider y being THING such that
                   8: In[y,A] & Iso[y,y'] by 7;

                   9: In[x,y,R] iff In[x',y',S] by 4,6,8,ISO;
                   10: In[x,y,R] by 3,9;
                   11: In[y,x,R] by 1,10,SymmetryDef;
            thus In[y',x',S] by 4,6,8,11,ISO;
        end;
        thus thesis by 2,SymmetryDef;
    end;
end;

thus
Exercise9:
Transitive[S] implies Transitive[R] proof
```

```
    assume 1: Transitive[S];
    thus Transitive[R] proof
        2: for x,y,z holds In[x,y,R] & In[y,z,R] implies In[x,z,R]
           proof
               let x,y,z be THING;
               assume 3: In[x,y,R] & In[y,z,R];
                      4: In[x,A] & In[y,A] & In[z,A] by 3,R1;

                      5: ex x' st In[x',B] & Iso[x,x'] by 4,T;
                      consider x' being THING such that
                      6: In[x',B] & Iso[x,x'] by 5;

                      7: ex x' st In[x',B] & Iso[y,x'] by 4,T;
                      consider y' being THING such that
                      8: In[y',B] & Iso[y,y'] by 7;

                      9: ex x' st In[x',B] & Iso[z,x'] by 4,T;
                      consider z' being THING such that
                      10: In[z',B] & Iso[z,z'] by 9;

                      11: In[x',y',S] & In[y',z',S] by 3,4,6,8,10,ISO;
                      12: In[x',z',S] by 1,11,TransitivityDef;
               thus In[x,z,R] by 4,6,10,12,ISO;
           end;
           thus thesis by 2,TransitivityDef;
    end;
end;

thus
Exercise10:
antiSymmetric[R] implies antiSymmetric[S] proof

    assume 1: antiSymmetric[R];
    thus antiSymmetric[S] proof
        2: for x,y holds In[x,y,S] & In[y,x,S] implies x=y proof

            let x',y' be THING;
            assume 3: In[x',y',S] & In[y',x',S];
                   4: In[x',B] & In[y',B] by 3,S1;

                   5: ex x st In[x,A] & Iso[x,x'] by 4,S;
                   consider x being THING such that
                   6: In[x,A] & Iso[x,x'] by 5,S;

                   7: ex x st In[x,A] & Iso[x,y'] by 4,S;
                   consider y being THING such that
                   8: In[y,A] & Iso[y,y'] by 7;

                   9: In[x,y,R] & In[y,x,R] by 3,4,6,8,ISO;
                   10: x=y by 1,9,AntiSymmetryDef;
```

```
                    11: Iso[x,y'] by 8,10;
                thus x'=y' by 4,6,11,F;
            end;
            thus thesis by 2,AntiSymmetryDef;
        end;
    end;

    end;
```

11.6 Relationships between Functions and Relations

Recall that $f = \langle D, R, G\rangle$ is a function iff the subset G of $D \times R$ has the property that for every $x \in D$ there is *at most* one $y \in R$ such that $\langle x, y\rangle \in G$.

Suppose you have a function $f = \langle X, Y, G\rangle$ Then this function *induces* a binary relation $r = \langle X \times Y, G\rangle$. That is, the ordered pairs of the relation r are exactly the ordered pairs $\langle x, f(x)\rangle$ where $x \in \text{domain}(f)$.

Example: Let $f = \langle X, Y, G\rangle$ where

$$\begin{aligned}X &= \{0, 1, 2, 3\}\\ Y &= \{0, 1, 2, \ldots, 9\}\\ G &= \{\langle 0, 0\rangle, \langle 1, 1\rangle, \langle 2, 4\rangle, \langle 3, 9\rangle\}\end{aligned}$$

Or giving G as a table:

x	$f(x)$
0	0
1	1
2	4
3	9

Note that a function *always* induces an associated binary relation.

Now suppose you have a binary relation $r = \langle D, G\rangle$ where $D = X \times Y$ and $G \subseteq D$. Then r induces a function f if r has the property that for every pair $\langle x, y\rangle \in G$ no other pair exists with the same x value. The function f induced by r is then given by $f = \langle X, Y, G\rangle$.

Because of this relationship between functions and their induced relations, and vice versa, functions are often defined as special kinds of binary relations.

Let's consider the case where the function f is 1–1. The induced relation $r = \langle X \times Y, G\rangle$ has the following property: For every ordered pair $\langle x, y\rangle \in G$ no other pairs exist with the same x or the same y value.

Now consider a new relation r^{-1} which comes from r by simply exchanging the x, y components of every ordered pair in G.

$$\begin{aligned}r^{-1} &= \langle Y \times X, G^{-1}\rangle\\ G^{-1} &= \{\langle y, x\rangle | \langle x, y\rangle \in G\}\end{aligned}$$

Since f is 1–1, every pair $\langle x, y\rangle \in G$ has a unique y value (that is, no other pair has the same y value). This means that every pair $\langle y, x\rangle \in G^{-1}$ has a unique x value. This means that r^{-1} induces a function, say f^{-1}, from Y to X given by $f^{-1} = \langle Y, X, G^{-1}\rangle$ with

G^{-1} :

y	$f(y)$
0	0
1	1
4	2
9	3

The original function f and the new function f^{-1} have the following properties:

1. f, f^{-1} are both 1–1
2. range(f) = domain(f^{-1})
3. domain(f) = range(f^{-1})
4. for each $x \in$ domain(f) we have $f^{-1}(f(x)) = x$
5. for each $y \in$ range(f) we have $f(f^{-1}(y)) = y$

Now in general, f^{-1} will be a partial function because range(f) is not necessarily all of Y. But if f is onto, then range(f) $= Y$, and so domain(f^{-1}) $= Y$, and f^{-1} is total.

Similarly, if f is total, then domain(f) $= X$, so range(f^{-1}) $= X$, and f^{-1} will be onto. So to the properties above we can add:

1. f is onto Y iff f^{-1} is total on Y
2. f is total on X iff f^{-1} is onto X

11.7 Variations on Invertibility

Now what about the case where f is not 1–1? Is it still possible to have a notion of inverse? This is equivalent to asking: if one has a binary relation $r = \langle D, G\rangle$ that lacks the property that every ordered pair of G has a unique first component, can one associate a function to r?

Example: Consider the function $f = \langle X, Y, H\rangle$ given by

$$X = \{-4, -3, -2, -1, 0, 1, 2, 3, 4\}$$
$$Y = \{0, 1, 2, \ldots, 9\}$$

with graph H given by

H :

x	$f(x)$
-3	9
-2	4
-1	1
0	0
1	1
2	4
3	9

Note that domain$(f) = \{-3, -2, -1, 0, 1, 2, 3\}$ and range$(f) = \{0, 1, 4, 9\}$. The function f is not invertible since it is not 1–1.

The relation $r = \langle Y \times X, G \rangle$, where

$$G = \{\langle 0, 0\rangle, \langle 1, -1\rangle, \langle 1, +1\rangle, \langle 4, -2\rangle, \langle 4, 2\rangle, \langle 9, -3\rangle, \langle 9, 3\rangle\}$$

identifies the x values associated with each y value in range(f). It cannot induce a function since it lacks the property that no two ordered pairs in G share the same first value.

We now look at the subset z of X associated with each $y \in Y$, and define a new function g as follows:

$$g = \langle Y, \mathcal{P}(X), G' \rangle$$

where

$$G' = \Big\{\langle y, z\rangle | y \in Y \text{ and } z = \{x | \langle y, x\rangle \in G\}\Big\}$$

or, using f instead of r to define G'

$$G' = \Big\{\langle y, z\rangle | y \in Y \text{ and } z = \{x | f(x) = y\}\Big\}$$

So, for our example, $G' =$

$$\Big\{\langle 0, \{0\}\rangle, \langle 1, \{-1, 1\}\rangle, \langle 4, \{-2, 2\}\rangle, \langle 9, \{-3, 3\}\rangle, \langle 2, \emptyset\rangle, \langle 3, \emptyset\rangle, \langle 5, \emptyset\rangle, \langle 6, \emptyset\rangle, \langle 7, \emptyset\rangle, \langle 8, \emptyset\rangle\Big\}$$

Which we can express as a table:

G' :

y	$g(y)$
0	{0}
1	{1, -1}
2	Ø
3	Ø
4	{2, -2}
5	Ø
6	Ø
7	Ø
8	Ø
9	{-3, 3}

Note that g is a total function, and its relationship to r and f is expressed by the following statement: For every $y \in Y$, for every $x \in g(y)$ we have $f(x) = y$ and $\langle y, x\rangle \in G$.

12 Induction

12.1 Inductive Construction, Definition, and Proofs

The natural numbers

$$Nat = \{0, 1, 2, \ldots\}$$

are a very important set. This set is infinite, yet everyone knows exactly what it contains. How can this be?

The reason that we feel that we know what Nat contains is that we know how to "construct" it. Starting with 0 being in Nat, we add 1 and place the result into Nat. Then we add 1 to this new element and place it (i.e. 2) into Nat. We keep doing this. Any element of Nat can eventually be constructed by this process of selecting an element from Nat, adding 1 to it, and putting the result into Nat. Here is a more precise description of this process:

Base Case: $0 \in Nat$

Induction Step: `for` $n \in Nat$ `holds` $n + 1 \in Nat$

Extremal Clause: Nat contains no other elements than those constructed above.

This is an *inductive definition* of the set Nat of natural numbers.

An inductive definition can apply to more than just sets of numbers. Consider character strings with the concatenation operator $\cdot$. Here is a definition of a set of formulas over the variables x, y, z using the operators negation, addition, and multiplication.

Base Case: "x", "y", "z" $\in F$

Induction Step: `for` $f, g \in F$ `holds` "$-$"$\cdot f \in F$ `&` "("$\cdot f \cdot$"+"$\cdot g \cdot$")"$\in F$ `&` "("$\cdot f \cdot$"$\times$"$\cdot g \cdot$")"$\in F$

The BNF grammar of Mizar is also an inductive definition of all the syntactically correct Mizar texts.

The general form of an inductive definition of a set A follows this pattern:

Base Case: Provide some fixed set $B = \{b_1, \ldots, b_k\}$ of basis elements to be included in A, that is $B \subset A$.

Induction Step: Provide a way of taking an arbitrary set $I = \{a_1, \ldots, a_l\}$ of elements already in A, and constructing a set $S = \{s_1, \ldots, s_m\}$ of elements to be added to A.

Extremal Clause: Specify that A contains no other elements than those constructed above.

The inductive definition of a set gives us a mechanism for proving that a given property holds for every member of the set. The idea is to show that some property P holds for the basis elements (show $P[0]$ in the case of Nat), and then to show that P is inherited by objects during the construction step (show that if $P[n]$ holds then $P[n+1]$ holds).

The general method of showing that a property P holds for all elements of an inductively defined set A, that is,

`for` $x \in A$ `holds` $P[x]$

follows this pattern:

Base Case: Prove that all objects in the basis set $B = \{b_1, \dots, b_k\}$ satisfy property P.

Induction Step: Show that if P holds for every element of an arbitrary subset $I = \{a_1, \dots, a_l\}$ of A, then P also holds for every element of the set $S = \{s_1, \dots, s_m\}$ containing all objects constructed from elements of I.

For example, here is a typical induction proof of the type one sees in high school.

```
for $n$ being $Nat$ holds $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$
proof

    BaseCase:  == when $n = 0$
            $\sum_{i=0}^{0} i = \frac{0(0+1)}{2}$

    InductionStep:  now
        let $n$ be $Nat$; == $n$ is an arbitrary $Nat$
        == assume that what we want to prove is true for $n$
        assume IndHyp: $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$
        == show that what we want to prove is true for $n+1$
        1: $(\sum_{i=0}^{n} i) + (n+1) = \frac{n(n+1)}{2} + n + 1$ by IndHyp;
        2: $\sum_{i=0}^{n+1} i = \frac{n(n+1)+2(n+1)}{2}$ by 1;
        thus $\sum_{i=0}^{n+1} i = \frac{(n+1)((n+1)+1)}{2}$ by 2;
    end;

    thus for $n$ being $Nat$ holds $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$ by BaseCase, InductionStep;
end;
```

Why should the conclusion follow from `BaseCase` and `InductionStep`? Surely, some kind of inference rule is required to justify the conclusion. Consider what the induction step actually shows:

```
for $n$ being $Nat$ holds $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$ implies $\sum_{i=0}^{n+1} i = \frac{(n+1)((n+1)+1)}{2}$
```

Thus the actual inference rule for induction must mention passing from a statement about n to the same statement about $n+1$, independently of n.

12.2 Natural Number Induction or Weak Induction

Let $Nat = \{0, 1, 2, 3, \ldots\}$ be the set of natural numbers. Let P be a 1-ary predicate defined on Nat. If the following two properties hold:

```
BaseCase: $P[0]$
InductionStep: for $n$ being $Nat$ holds $P[n]$ implies $P[n+1]$
```

then you can conclude

```
for $n$ being $Nat$ holds $P[n]$
```

Note: for a particular predicate P the axiom of natural number induction can be stated as the single logical formula:

```
P[0] & (for $n$ being $Nat$ holds $P[n]$ implies $P[n+1]$)
        implies
(for $n$ being $Nat$ holds $P[n]$)
```

We will refer to this rule of inference as `NaturalNumberInduction`.

Here are some examples of inductive proofs of the kind you typically see in high school. Note how we have used the Mizar proof style, event though Mizar does not understand arithmetic.

12.2.1 A geometric sum

Let $P[n]$ be the predicate on natural numbers that is true when

$$3^{n+1} - 1 = \sum_{i=0}^{i=n} 2 \cdot 3^i$$

```
for $n$ being $Nat$ holds $P[n]$
proof
    BaseCase: $P[0]$
    proof
            $3^{0+1} - 1 = 2 \cdot 3^0$
        thus thesis
    end;
    InductionStep: for $n$ being $Nat$ holds $P[n]$ implies $P[n+1]$
    proof
        let $n$ be $Nat$;
        assume $P[n]$;
            $3^{n+1} - 1 = \sum_{i=0}^{i=n} 2 \cdot 3^i$;
            $3^{n+1} - 1 + 2 \cdot 3^{n+1} = 2 \cdot 3^{n+1} + \sum_{i=0}^{i=n} 2 \cdot 3^i$;
            $3 \cdot 3^{n+1} - 1 = \sum_{i=0}^{i=n+1} 2 \cdot 3^i$;
        thus $P[n+1]$;
    end;
    thus thesis by BaseCase, InductionStep, NaturalNumberInduction;
end;
```

12.2.2 A quadratic sum

Find a closed form [5] formula for:

$$0^2 + 1^2 + 2^2 + 3^2 + \cdots + k^2$$

which using the Σ operator is written as:

$$\sum_{i=0}^{i=k} i^2$$

Note: The Principle of Induction, or any of what we call logic for that matter, does not tell us how to guess the closed formula. However, we are free to make a very intelligent guess that for arbitrary k:

$$\sum_{i=0}^{i=k} i^2 = \frac{1}{6}k(k+1)(2k+1)$$

Now using the Principle of Induction we can check whether or not our intelligent guess was correct. What are we proving?

$$\texttt{for } k \texttt{ being } Nat \texttt{ holds } (\sum_{i=0}^{i=k} i^2 = \frac{1}{6}k(k+1)(2k+1))$$

Which has the shape of " `for` k `being` Nat `holds` $P[k]$ " where

$$P[k] \quad \texttt{iff} \quad \sum_{i=0}^{i=k} i^2 = \frac{1}{6}k(k+1)(2k+1)$$

```
for $k$ being $Nat$ holds $P[k]$
proof
    BaseCase: $P[0]$
    proof
            $\sum_{i=0}^{i=0} i^2 = \frac{1}{6}0(0+1)(2 \cdot 0+1)$;
        thus thesis
    end;

    InductionStep: for $k$ being $Nat$ holds $P[k]$ implies $P[k+1]$
    proof
            == Note that the current thesis is:
            ==  for $k$ being $Nat$ holds ($\sum_{i=0}^{i=k} i^2 = \frac{1}{6}k(k+1)(2k+1)$
            ==                        implies
            ==        $\sum_{i=0}^{i=k+1} i^2 = \frac{1}{6}(k+1)((k+1)+1)(2(k+1)+1)$)
            == What are we to prove?
            == Recall the Universal Quantifier Introduction rule.
        let $n$ be $Nat$;                         == An *arbitrary* natural number.
```

[5] We say that a formula is in a closed form if it involves only a fixed number of simple operations. The summation operator hides a variable number of additions, and thus a formula using it is not closed.

```
                == What remains to be proven?
                == Recall The Implication Introduction rule.
            assume InductionHypothesis: $\sum_{i=0}^{i=n} i^2 = \frac{1}{6}n(n+1)(2n+1)$;
                == The Principle of Induction does not give us a hint how to get
                == the consequent.
                == Take $lhs$ of the consequent and "massage" it.
        $\sum_{i=0}^{i=n+1} i^2$  $= \sum_{i=0}^{i=n} i^2 + (n+1)^2$                  == isolate $n+1$ term
                  $= \frac{1}{6}n(n+1)(2n+1) + (n+1)^2$          == from InductionHypothesis
                  $= \frac{1}{6}(n+1)(n(2n+1) + 6(n+1))$         == factor out $n+1$
                  $= \frac{1}{6}(n+1)(2n^2 + n + 6n + 6)$                    == multiply
                  $= \frac{1}{6}(n+1)(n(2n+3) + 2(2n+3))$                    == regroup
                  $= \frac{1}{6}(n+1)(n+2)(2n+3)$                          == factor out
                  $= \frac{1}{6}(n+1)((n+1)+1)(2(n+1)+1)$;              == and regroup
                == and we have the $rhs$ of the consequent.
            thus thesis
        end;
        thus for $k$ being $Nat$ holds $(\sum_{i=0}^{i=k} i^2 = \frac{1}{6}k(k+1)(2k+1))$
                         by BaseCase, InductionStep, NaturalNumberInduction
    end
```

12.2.3 A confusing example

Find a closed form formula for:

$$\sum_{i=0}^{i=k} 1$$

We make an intelligent guess that for arbitrary k:

$$\sum_{i=0}^{i=k} 1 = (k+1)$$

which can be proven by induction. What are we proving?

```
    for $k$ being $Nat$ holds $\sum_{i=0}^{i=k} 1 = (k+1)$
    proof
            == The thesis has the shape of for $k$ being $Nat$ holds $P[k]$
            == where $P[k]$ iff $\sum_{i=0}^{i=k} 1 = (k+1)$.
        BaseCase: $\sum_{i=0}^{i=0} 1 = 0 + 1$;                 == i.e. $P[0]$, is obviously true.
        InductionStep: for $k$ being $Nat$ holds $P[k]$ implies $P[k+1]$
        proof
            let $k$ be $Nat$;                                    == $k$ is an arbitrary $Nat$.
            assume InductionHypothesis: $\sum_{i=0}^{i=k} 1 = k + 1$;

                $\sum_{i=0}^{i=k+1} 1 = \left(\sum_{i=0}^{i=k} 1\right) + 1$
                           $= (k + 1) + 1$        == by InductionHypothesis.

            thus thesis
        end
        thus thesis by BaseCase, InductionStep, NaturalNumberInduction
    end
```

12.2.4 Programming examples

Here is another, more "realistic" example of induction use. Consider the following function (we hope you can follow the notation):

```
function $S(x) : Nat \mapsto Nat$
    pre: $x \geq 0$
    post: $S(x) = x^2$

    if $x = 0$ then
        return 0;
    else
        return $S(x - 1) + 2 * x - 1$;
    endif;
end;
```

Observe how $S(x)$ is expressed recursively in terms of $S(x - 1)$ when $x > 0$.

```
Claim: for $x$ being $Nat$ holds $S(x) = x^2$
proof
    BaseCase: $S(0) = 0$ by definition of $S$;
    InductionStep: for $x$ being $Nat$ holds $S(x) = x^2$ implies $S(x+1) = (x+1)^2$

    proof
        let $x$ be $Nat$;
        assume IndHyp: $S(x) = x^2$;
        1: $x + 1 > 0$ by ?;
        2: $S(x + 1) = S(x) + 2(x + 1) - 1$ by 1, definition of $S$;
        3: $S(x + 1) = x^2 + 2x + 1$ by 2, IndHyp, ?;
        thus $S(x + 1) = (x + 1)^2$ by 3;
    end;
    thus thesis by BaseCase, InductionStep, NaturalNumberInduction
end;
```

Question: What would happen if we used *Int* instead of *Nat* and we attempted to evaluate $S(x)$ for $x < 0$?

Here is a another recursively defined function.

```
function $M(x, y) : Nat \times Nat \mapsto Nat$
    pre: *none.*
    post: $M(x, y) = x \cdot y$.

    if $x = 0$ then
        return 0;
    else
        return $y + M(x - 1, y)$;
    endif;
end;
```

What does this program do?

```
Claim: for $x, y$ being $Nat$ holds $M(x, y) = xy$
proof
    == Induction on $x$;
    BaseCase: for $y$ being $Nat$ holds $M(0, y) = 0$ by definition of $M$;
    InductionStep: for $x$ being $Nat$ holds
        (for $y$ being $Nat$ holds $M(x, y) = xy$)
    implies
        (for $y$ being $Nat$ holds $M(x + 1, y) = (x + 1)y$)
    proof
        let $x$ be $Nat$;
        assume IndHyp: for $y$ being $Nat$ holds $M(x, y) = xy$
        0:  now
            let $y$ be $Nat$;
            1: $x + 1 > 0$ by ?;
            2: $M(x + 1, y) = y + M(x, y)$ by 1, definition of $M$;
            3: $M(x + 1, y) = y + xy$ by 2, IndHyp, ?;
            thus $M(x + 1, y) = y(x + 1)$ by 3;
        end;
        thus for $y$ being $Nat$ holds $M(x + 1, y) = (x + 1)y$ by 0;
    end;
    thus thesis by BaseCase, InductionStep, NaturalNumberInduction
end;
```

Question: what would happen if we changed the type of y to *Int*? Does the above proof still work? What if the type of x is changed to *Int*? In this case we could define another function to *reduce* integer multiplication to natural multiplication:

```
function $IM(x, y) : Int \times Int \mapsto Int$
    if $x < 0$ then
        return $-M(-x, y)$;
    else
        return $M(x, y)$;
    endif;
end;
```

Question: is the type casting done above to change the sign of x and the result legal?

12.2.5 Fibonacci sequence

The original problem that was stated by Fibonacci (Leonardo of Pisa) was as follows. A pair of guinea pigs born at time 1 creates a further pair in every month of its existence, beginning with the second month, and its descendants behave the same way. How many pairs will there be after n months.

Let $F : N \mapsto N$ and $F(n)$ be the number of pairs after n months. The graph of F is inductively defined as (we assume that after 0 months there were 0 pairs)

1° (basis) $\langle 0, 0\rangle \in F$, $\langle 1, 1\rangle \in F$.
2° If $\langle k, x\rangle \in F$ and $\langle k+1, y\rangle \in F$ then $\langle k+2, x+y\rangle \in F$, for any $k \in N$.
3° Nothing else is in F but what was mentioned in 1° and 2°.

Exercise: Construct first 10 elements of F.

We will use the more common notation for sequences, instead of $\langle x, y\rangle \in F$ we write $F_x = y$.

With this notation F_n is defined *recursively* as:
1° (stop condition) $F_0 = 0$, $F_1 = 1$.
2° (recursion) $F_n = F_{n-1} + F_{n-2}$, for $n \geq 2$.

The two definitions above define exactly the same thing.

Prove that for every $n \in Nat$ the following property $P[n]$ holds

$$F_n \leq \left(\frac{1+\sqrt{5}}{2}\right)^{n-1}$$

We will use the following notation: $\varphi = \left(\frac{1+\sqrt{5}}{2}\right)$. What are we proving?

```
for $n$ being $Nat$ holds $F_n \leq \left(\frac{1+\sqrt{5}}{2}\right)^{n-1}$
proof

    BaseCase:                                              == Show $P[0]$, i.e.
        $F_0 \leq \left(\frac{1+\sqrt{5}}{2}\right)^{0-1}$          == since $F_0 = 0$ and the $rhs$ = $\left(\frac{2}{1+\sqrt{5}}\right)$
    InductionStep: for $n$ being $Nat$ holds $P[n]$ implies $P[n+1]$
    proof
        let $n$ being $Nat$;                               == an arbitrary natural
        assume $F_n \leq \varphi^{n-1}$;
```

At this point we are to show $P[n+1]$, that is, $F_{n+1} \leq \varphi^{(n+1)-1}$. We know that $F_{n+1} = F_n + F_{n-1}$. But we know *nothing* specific about F_{n-1}! We get stuck and cannot continue.

However, there is a remedy in this case. Surprisingly, proving a stronger proposition is easier. We will prove:

```
for $n$ being $Nat$ holds ($F_n \leq \varphi^{n-1}$ & $F_{n+1} \leq \varphi^n$)
proof
```

```
                                    == from which the original proposition easily follows.
    BaseCase: $0 \leq \varphi^{0-1}$ & $1 \leq \varphi^0$;                  == which is easy to see

    InductionStep: now
        let $k$ be $Nat$;
        assume InductionHypothesis: $F_k \leq \varphi^{k-1}$ & $F_{k+1} \leq \varphi^k$;
        thus $F_{k+1} \leq \varphi^k$;              == The first part of what we want to prove.

            == the second conjunct, $F_{k+2} \leq \varphi^{k+1}$ is proven as follows:
```

$$\begin{array}{rcll} F_{k+2} & = & F_{k+1} + F_k & \text{== see definition of } F \\ & \leq & \varphi^k + \varphi^{k-1} & \text{== by InductionHypothesis} \\ & = & (\varphi+1)\varphi^{k-1} & \text{== factor out} \\ & = & \varphi^2\varphi^{k-1} & \text{== check that } \varphi + 1 = \varphi^2 \\ & = & \varphi^{k+1}; & \end{array}$$

```
        thus $F_{k+2} \leq \varphi^{k+1}$;        == The second part of what we want to prove.
    end
    thus thesis
end
```

Where did we pick up $\varphi = \left(\frac{1+\sqrt{5}}{2}\right)$? Answer: Because we are extremely clever.

But people were that clever a long time ago: Fibonacci—Leonardo of Pisa studied the sequence around 1202. The ancient Greeks knew about φ even earlier—they considered the ratio φ to 1 to be the most pleasing aesthetically, calling it the “Golden Ratio”.

Exercise. Prove that for all $n \in N$

$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

The above formula did not come out of the blue. The theory of *difference equations* offers a number of methods for finding closed formulae for recursively defined functions. However, this theory is generally considered quite advanced.

12.2.6 Powersets

Given set A, $|A|$ is the cardinality of A and $\mathcal{P}(A)$ is the powerset of A. We will try to prove the following theorem:

for n being Nat holds for A being Set holds $|A| = n$ implies $|\mathcal{P}(A)| = 2^n$.

We will save on writing by using an alternative Mizar syntax for the above sentence:

for n being Nat, A being Set st $|A| = n$ holds $|\mathcal{P}(A)| = 2^n$.

First, notice that the property of natural numbers $P[n]$ that we are trying to prove here is:

```
        P[n]   iff   (for A being Set holds |A| = n implies |P(A)| = 2^n).

    for n being Nat, A being Set st |A| = n holds |P(A)| = 2^n
    proof
            == By induction over n.

        BaseCase: for A being Set st |A| = 0 holds |P(A)| = 2^0
        proof
            == Do you now how to prove it?
            == Hint: if |B| = 0 then B = Ø and |P(Ø)| = 1.
        end;

        InductionStep: for n being Nat holds P[n] implies P[n+1]
        proof
            let n be Nat;
            assume InductionHypothesis:
                                   for A being Set st |A| = n holds |P(A)| = 2^n;
            thus for A being Set st |A| = n+1 holds |P(A)| = 2^(n+1)
            proof
                == Sketch of the proof. Let A be a set with n+1 elements

                                   A = {a_1, a_2, a_3, ... a_n, a_(n+1)}

                == we can look at its powerset as:

                       P(A) =  {B | B ∈ P(A \ {a_(n+1)})}  ∪
                                  {B ∪ {a_(n+1)} | B ∈ P(A \ {a_(n+1)})}

                == Because of |A \ {a_(n+1)}| = n then by InductionHypothesis

                                     |P(A \ {a_(n+1)})| = 2^n.

                == Therefore

                       |P(A)|  =  |{B | B ∈ P(A \ {a_(n+1)})}|  +
                               =       |{B ∪ {a_(n+1)} | B ∈ P(A \ {a_(n+1)})}|
                               =  2^n + 2^n = 2^(n+1).

                thus thesis
            end
        end
    end
```

12.2.7 Factoring polynomials

Prove that $(x - y)$ is a factor of $(x^n - y^n)$, for all $n \geq 1$.

We will write $P \mid R$ to say that polynomial P is a factor of polynomial R. What do we mean by $P \mid R$? Here is the definition:

```
for $P, R$ being *polynomial* holds
$P \mid R$  iff  ex $S$ being *polynomial* st $R = P \times S$.
```

What are we proving?

```
for $n$ being $Nat$ holds ($n \geq 1$ implies $(x - y) \mid (x^n - y^n)$).
```

What property of natural numbers do we prove? Here it is:

```
$P[n]$ iff ($n \geq 1$ implies $(x - y) \mid (x^n - y^n)$).
```

```
for $n$ being $Nat$ holds ($n \geq 1$ implies $(x - y) \mid (x^n - y^n)$)
proof
    BaseCase: $0 \geq 1$ implies $(x - y) \mid (x^0 - y^0)$;          == vacuously true
    InductionStep: for $k$ being $Nat$ holds $P[k]$ implies $P[k + 1]$
    proof
        let $n$ be $Nat$;                               == an arbitrary natural number
        assume InductionHypothesis: $n \geq 1$ implies $(x - y) \mid (x^n - y^n)$;
            == What remains to be proven?
        assume $(n + 1) \geq 1$;                                    == antecedent
            == Note that two cases are possible: $(n + 1) = 1$ or $(n + 1) > 1$.
        Case1: now
            assume $n + 1 = 1$;
                $n = 0$;
            thus $(x - y) \mid (x^{0+1} - y^{0+1})$
        end;
        Case2: now
            assume $n + 1 > 1$;
                $n \geq 1$;
                $(x - y) \mid (x^n - y^n)$ by InductionHypothesis;
                consider $S$ being *polynomial* such that
            1: $(x - y) \times S = (x^n - y^n)$;                 == by definition of |.
            == To prove the consequent, take $x^{n+1} - y^{n+1}$ and “massage” it.
                $x^{n+1} - y^{n+1} = xx^n - yy^n$
                              $= x((x - y)S + y^n) - yy^n$        == from 1
                              $= x(x - y)S + xy^n - yy^n$
                              $= x(x - y)S + (x - y)y^n$
                              $= (x - y)(xS + y^n)$;
            thus $(x - y) \mid (x^{n+1} - y^{n+1})$;                  == Do you see why?
        end
        thus thesis by Case1, Case2
    end
    thus for $n$ being $Nat$ holds ($n \geq 1$ implies $(x - y) \mid (x^n - y^n)$)
                    by BaseCase, InductionStep, NaturalNumberInduction
end
```

12.3 Integer Induction

There is nothing special about the set $Nat = \{0, 1, 2, \ldots\}$ except for two properties:

1. There is a first element, 0.
2. There is a way of moving from one element x to the next one, $x + 1$, without missing any in between.

These properties are also true for any set of integers of the form:

$$A = \{n | n \in Int \text{ and } n \geq a\}$$

for some constant a. (Alternatively, $A = \{a, a+1, a+2, \ldots\}$.) Thus we expect the following induction rule to be legitimate:

```
P[a] & (for $n$ being $Int$ st $n \geq a$ holds $P[n]$ implies $P[n+1]$)
        implies
(for $n$ being $Int$ st $n \geq a$ holds $P[n]$)
```

Note how this is like the axiom for natural number induction except that an additional guard of $n \geq a$ has been placed on n. This can also be expressed as:

Let a be an integer, and let P be a 1-ary predicate defined on the subset $A = \{n | n \in Int \text{ and } n \geq a\}$. If the following two properties hold:

```
BaseCase: $P[a]$
InductionStep: for $n$ being $Int$ st $n \in A$ holds $P[n]$ implies $P[n+1]$
```

then you can conclude

```
for $n$ being $Int$ st $n \in A$ holds $P[n]$
```

When $a = 0$ this gives us the axiom for natural number induction.

12.3.1 Factoring polynomials again

We will now redo the example 12.2.7 using the integer induction in place of the natural number induction. Let Pos be the set of positive integers, $Pos = \{1, 2, 3, \ldots\}$. The property of elements in Pos that we are proving now is:

$$P[n] \texttt{ iff } (x - y) \mid (x^n - y^n).$$

What are we proving now?

```
for $n$ being $Pos$ holds $(x-y) \mid (x^n - y^n)$
proof
    BaseCase: $(x-y) \mid (x^1 - y^1)$;                    == obviously true
        == Note the difference from the previous proof.
    InductionStep: for $k$ being $Pos$ holds $P[k]$ implies $P[k+1]$
    proof
```

```
        let $n$ be $Pos$;                          == An arbitrary positive integer.
        assume InductionHypothesis: $(x-y) \mid (x^n - y^n)$;

            == What remains to be proven? $(x-y) \mid (x^{n+1} - y^{n+1})$;
            == We will construct the polynomial required to prove this.

        consider $S$ being em polynomial such that
            1: $(x-y) \times S = (x^n - y^n)$;
                                   == by definition of | and InductionHypothesis.

        == To prove the consequent, take $x^{n+1} - y^{n+1}$ and "massage" it.
```

$$\begin{array}{rcll} x^{n+1} - y^{n+1} & = & xx^n - yy^n & \\ & = & x((x-y)S + y^n) - yy^n & \texttt{==}\text{ by } \texttt{1} \\ & = & x(x-y)S + xy^n - yy^n & \\ & = & x(x-y)S + (x-y)y^n & \\ & = & (x-y)(xS + y^n) & \end{array}$$

```
        thus $(x-y) \mid (x^{n+1} - y^{n+1})$          == Make sure you understand this.
    end
    thus for $n$ being $Pos$ holds $(x-y) \mid (x^n - y^n)$
                                by BaseCase, InductionStep, IntegerInduction
end
```

12.4 Complete Induction

Consider this program:

```
function $Fib(i) : Nat \mapsto Nat$
    pre: $i \geq 0$
    post: $Fib(i)$ is the $i$'th Fibonacci number

    if $i = 0$ then
        return 0;
    else if $i = 1$ then
        return 1;
    else
        return $Fib(i-2) + Fib(i-1)$;
    endif;
end;
```

This recursion requires two evaluations of Fib and so does not easily fit into the induction patterns discussed so far. But we have seen how to solve this problem (cf. 12.2.5).

A similar problem occurs in the typical tree traversal:

```
traverse(T):
    traverse(left-subtree(T))
    do root(T)
    traverse(right-subtree(T))
```

These kinds of problems, in which a given instance is broken into many smaller problems, leads to the notion of complete induction. In its version for natural numbers it is:

(`for` n `being` Nat `holds` (`for` m `being` Nat `st` $m < n$ `holds` $P[m]$) `implies` $P[n]$)
`implies`
(`for` n `being` Nat `holds` $P[n]$)

The corresponding complete integer induction is:

Let a be an integer, and let P be a 1-ary predicate defined on the subset of integers $A = \{n | n \in Int \text{ and } n \geq a\}$. If the following property holds:

`for` n being Int `st` $n \in A$ `holds`
((`for` m `being` Int `st` $m \in A$ `&` $m < n$ `holds` $P[m]$) `implies` $P[n]$)

then you can conclude

`for` n `being` Int `st` $n \in A$ `holds` $P[n]$

Here is an example of complete natural number induction:
`given` $\varphi = \frac{1+\sqrt{5}}{2}$
`Lemma:` $1 + \varphi = \varphi^2$; (An easy proof is left as an exercise to the reader).

```
Claim: for $n$ being $Nat$ st $n \geq 0$ holds $Fib(n) \leq \varphi^{n-1}$ proof
    0:  now
        let $n$ be $Nat$;
        assume IndHyp: for $m$ being $Nat$ st $m < n$ holds $Fib(m) \leq \varphi^{m-1}$
        1: $n < 2$ or $2 \leq n$ by ?;
        2: $n < 2$ implies $Fib(n) \leq \varphi^{n-1}$
        proof
            assume A: $n < 2$;
            2a: $n = 0$ or $n = 1$ by A;
            2b: $Fib(n) = n$ by 2a, definition of $Fib$;
            2c: $0 \leq \varphi^{-1}$;
            2d: $1 \leq \varphi^{0}$;
            thus $Fib(n) \leq \varphi^{n-1}$ by 2a, 2b, 2c, 2d;
        end;
        3: $n \geq 2$ implies $Fib(n) \leq \varphi^{n-1}$
        proof
            assume B: $n \geq 2$;
            3a: $0 \leq n-2$ & $0 \leq n-1$ by B;
            3b: $Fib(n) = Fib(n-2) + Fib(n-1)$ by 3a, definition of $Fib$;
            3c: $Fib(n-2) \leq \varphi^{n-3}$ & $Fib(n-1) \leq \varphi^{n-2}$ by B, IndHyp;
            3d: $Fib(n) \leq \varphi^{n-3} + \varphi^{n-2}$ by 3b, 3c;
            3e: $Fib(n) \leq \varphi^{n-3}(1+\varphi)$ by 3d;
            thus $Fib(n) \leq \varphi^{n-1}$ by 3e, Lemma;
        end;
        thus $Fib(n) \leq \varphi^{n-1}$ by 1, 2, 3;
    end;
    thus thesis by 0, CompleteInduction;
end;
```

Note: make sure that you see what is proven by the reasoning labelled 0.

12.4.1 An example of complete induction

We have a function $T : N \mapsto N$ defined as:

$$T(n) = \begin{cases} T(\lfloor n/5 \rfloor) + T(\lfloor 3n/4 \rfloor) + n & \text{if } n > 0 \\ 0 & n = 0 \end{cases}$$

(The *floor* of r, $\lfloor r \rfloor$, for real r is the largest integer not greater than r.)

Similar functions are common in the analysis of algorithms that use Divide and Conquer. Note the trouble with weak induction when proving

for n being Nat holds $T(n) \leq 20n$.

Where is the trouble? Not with the base case. If you start the proof of the induction step then you will quickly notice that the inductive hypothesis is essentially useless.

Therefore we will proceed by complete induction. The property of natural numbers that we are concerned here with is:

$$P[n] \quad \texttt{iff} \quad T(n) \leq 20n.$$

and to apply complete induction we have to prove the following single statement:

```
for $n$ being $Nat$ holds
    (for $k$ being $Nat$ st $k < n$ holds $P[k]$) implies $P[n]$.
```

which quite appropriately can be named an induction step.

```
InductionStep:
    for $n$ being $Nat$ holds
        (for $k$ being $Nat$ st $k < n$ holds $T(k) \leq 20k$) implies $T(n) \leq 20n$
    proof
        let $k$ be $Nat$;                                   == An arbitrary natural.
        assume InductionHypothesis:
                                   for $l$ being $Nat$ st $l < k$ holds $T(l) \leq 20l$;
        == We are to prove that $T(k) \leq 20k$.
        == Since $k$ is arbitrary, we will consider 2 cases: $k = 0$ or $k > 0$.

    Case1: now assume $k = 0$;
            == Note that the InductionHypothesis is useless.
            == We have to do the base case anyway. It is easy to see that
        thus $T(0) = 0 \leq 20 \cdot 0$
    end

    Case2: now
        assume $k > 0$;
        1: $T(k) = T(\lfloor k/5 \rfloor) + T(\lfloor 3k/4 \rfloor) + k$;          == by the definition of $T$
        2: $\lfloor k/5 \rfloor < k$ & $\lfloor 3k/4 \rfloor < k$;                     == simple arithmetic

          $T(k) \leq 20\lfloor k/5 \rfloor + 20\lfloor 3k/4 \rfloor + k$
                          by 1, 2, InductionHypothesis
                $\leq 20k/5 + 20 \cdot 3k/4 + k$                          as $\lfloor a \rfloor \leq a$
                $= 4k + 15k + k$
                $= 20k$

        thus $T(k) \leq 20k$
    end thus thesis by Case1, Case2
end
```

Therefore we conclude

```
for $n$ being $Nat$ holds $T(n) \leq 20n$ by InductionStep, CompleteInduction
```

12.4.2 Another Example of Complete Induction

Induction is particularly useful when reasoning about recursive functions. Here is a deceptively simple recursive definition of a function $F \mapsto Int \times Int$.

$$\texttt{DefFx}: \qquad F(x) = \begin{cases} x - 10 & \text{if } x > 100 \\ F(F(x+11)) & \text{if } x \leq 100 \end{cases}$$

In Pascal this could be written as

```
function F(x:  integer):  integer;
    if x > 100 then
        F := x-10;
    else
        F := F(F(x+11));
    endif;
end;
```

We wish to prove that for any integer x,

$$F(x) = \begin{cases} x - 10 & \text{if } x > 100 \\ 91 & \text{if } x \le 100 \end{cases}$$

Let us begin by defining two predicates that mirror the two cases in the program for F. Predicate H corresponds to the "then" part, and predicate E corresponds to the "else" part.

```
for $x \in Int$ holds
    $H[x]$ iff ( $x > 100$ implies $F(x) = x - 10$ )
for $x \in Int$ holds
    $E[x]$ iff ( $x \le 100$ implies $F(x) = 91$ )
```

What we want to prove is `for` $x \in Int$ `holds` $H[x]$ `&` $E[x]$.

Now, a simple inspection of the definition of F, `DefFx`, shows that `for` $x \in Int$ `holds` $H[x]$ is true. But a proof for the "else" case requires complete induction. Here is an informal proof in Mizar style.

```
for $x \in Int$ holds $x \le 100$ implies $F(x) = 91$ proof
    now
        let $x$ be $Int$ such that A1: $x \le 100$;
        assume IndHyp: for $m \in Int$ st $x < m \le 100$ holds $F(m) = 91$;
        thus $F(x) = 91$ proof
            Fx: $F(x) = F(F(x + 11))$ by A1, DefFx;
            Case1: now assume A2: $x + 11 \le 100$;
                S1: $x < x + 11 \le 100$ by A2;
                S2: $F(x + 11) = 91$ by S1, IndHyp ;
                S3: $x < 91 \le 100$ by A2 ;
                S4: $F(91) = 91$ by S3, IndHyp ;
                S5: $F(F(x + 11)) = 91$ by S2, S4;
                thus $F(x) = 91$ by S5, Fx;
            end;
            Case2: now assume A2: $x + 11 > 100$;
                S1: $F(x + 11) = x + 1$ by A2, DefFx;
                Case2a: now assume A3: $x + 1 = 101$;
                    thus $F(x + 1) = 91$ by A3, DefFx;
                end;
                Case2b: now assume A3: $x + 1 \le 100$;
                    thus $F(x + 1) = 91$ by A3, IndHyp;
                end;
                S2: $F(F(x+11)) = 91$ by A1, S1, Case2a, Case2b;
                thus $F(x) = 91$ by S2;
            end;
            thus $F(x) = 91$ by Case1, Case2;
        end;
    end;
    thus thesis by CompleteInduction
end;
```

12.4.3 A Variant of Complete Induction

This variant works for a subset of *Int* bounded from above. Note how each step constructs a smaller element of A.

Let a be an integer, and let P be a 1-ary predicate defined on the subset of integers $A = \{n | n \in Int \text{ and } n \leq a\}$. If the following property holds:

```
for $n$ being $Int$ st $n \in A$ holds
((for $m$ being $Int$ st $m \in A$ & $n < m$ holds $P[m]$) implies $P[n]$)
```

then you can conclude

```
for $n$ being $Int$ st $n \in A$ holds $P[n]$
```

12.4.4 Least Element Principle

The last induction scheme is not based on construction: Let A be a non-empty set of natural numbers. Then A has a smallest element. That is

`ex` n `being` Nat `st` $n \in A$ `&` (`for` m `being` Nat `st` $m \in A$ `holds` $n \leq m$)

The above induction axioms for natural numbers can be proven using the least element principle, and vice versa.

12.5 An incorrect inductive proof

Consider the following inductive proof.

Theorem: For all natural numbers n, for all sets of natural numbers A such that A contains n elements, it holds that all members of A are even.

Proof: We proceed by induction on the size of the set A.

Define, for any natural number n, the property $E[n]$ as follows: $E[n]$ iff for any set A of natural numbers such that A contains n members it holds that all members of A are even.

> Base Case: $E[0]$ is vacuously true since all elements of an empty set of natural numbers are even.
>
> Induction Step: Let n be an arbitrary natural number, suppose that $E[n]$ holds. We will show that $E[n+1]$ holds.
>
> Let A be any set of natural numbers of size $n+1$. Pick any two distinct elements of A, say x and y, and form two new sets $A_x = A - \{x\}$ and $A_y = A - \{y\}$ by removing x and y (respectively) from A. Since $x \in A_y$ and $y \in A_x$ we have $A = A_x \cup A_y$.
>
> Now both A_x and A_y have size n. Thus we can apply the induction hypothesis, $E[n]$, to conclude that all elements of A_x and A_y are even.
>
> But $A = A_x \cup A_y$, so we also have that all elements of set A are even. Thus $E[n+1]$ holds.

Thus, for any natural number n, holds $E[n]$, and the theorem is proven.

End proof.

The theorem is clearly false. What is wrong with this proof?

The problem is in the step that says "Pick any two distinct elements of A ...". This must be possible for any set of size $n+1$. But n can be 0, in which case $n+1$ is 1, and it is impossible to pick two distinct elements from a set of size 1.

Part IV
Collected Examples

A Sample Mizar Proofs

We used to have a bunch of Mizar proofs from the past assignments in this section. Now they are spread all over the notes.

A.1 A murder mystery

(agatha.mse)

```
== The following puzzle is due to L. Schubert
==
== A mysterious death at Dreadbury Mansion ...
==
==  1: Someone who lives at Dreadbury Mansion killed Aunt Agatha.
==  2: Only Agatha, the butler and Charles has lived at Dreadbury Mansion.
==  3: A killer always hates their victim.
==  4: A killer is never richer than their victim.
==  5: Charles hates no-one that Aunt Agatha hates.
==  6: Aunt Agatha hates everyone but the butler.
==  7: The butler hates everyone who is not richer than Aunt Agatha.
==  8: The butler hates everyone Aunt Agatha hates.
==  9: No-one hates everyone.
== 10: Aunt Agatha is not identical with the butler, of course.
==
== Question: Who killed Aunt Agatha?
==

environ reserve x1,x2,z,u,v,w for Person;
 given Agatha, butler, Charles being Person;
    1:  ex x1 st LivesatDBM[x1] & Killed[x1,Agatha];
    2:  for x1 st LivesatDBM[x1] holds x1=Agatha or x1=butler or x1=Charles;
    3:  for x1,x2 st Killed[x1,x2] holds Hates[x1,x2];
    4:  for x1,x2 st Killed[x1,x2] holds not Richer[x1,x2];
    5:  for x1 st Hates[Agatha,x1] holds not Hates[Charles,x1];
    6:  for x1 st x1<>butler holds Hates[Agatha,x1];
    7:  for x1 st not Richer[x1,Agatha] holds Hates[butler,x1];
    8:  for x1 st Hates[Agatha,x1] holds Hates[butler,x1];
    9: for x1 ex x2 st not Hates[x1,x2];
   10: Agatha<>butler

begin

 Killed[Agatha, Agatha]
   proof

       11: for x1 st x1<>butler holds Hates[butler,x1] proof
           let u be Person;
           assume 11a: u<>butler;
           thus Hates[butler,u] proof
               11b: Hates[Agatha,u] by 11a,6;
               thus thesis by 8,11b;
```

```
            end;
        end;

        12: ex x2 st not Hates[butler,x2] by 9;
        consider u such that
        13: not Hates[butler,u] by 12;

        14: u=butler proof
            assume 14a: u<>butler;
                   14b: Hates[butler,u] by 11,14a;
            thus contradiction by 13,14b;
        end;

        15: not Hates[butler,butler] by 13,14;
        16: Richer[butler,Agatha] by 7,15;
        17: not Killed[butler,Agatha] by 4,16;

        18: Hates[Agatha,Agatha] by 6,10;
        19: not Hates[Charles,Agatha] by 5,18;
        20: not Killed[Charles,Agatha] by 3,19;

        consider u such that
        21: LivesatDBM[u] & Killed[u,Agatha] by 1;
        22: u=Agatha or u=butler or u=Charles by 2,21;
        23: u<>butler & u<>Charles by 17,20,21;
        24: u=Agatha by 22,23;

     thus thesis by 21,24;

    end
```

A.2 K. Schubert's Steamroller

(steamr.mse)

```
environ

== The following example is due to L. Schubert.
==
== Wolves, foxes, birds, caterpillars, and snails are animals, and there
== are some of each of them.

 reserve x, y, w, f, b, c, s for animal;

A: ex w, f, b, c, s st
      wolf[w] & fox[f] & bird[b] & caterpillar[c] & snail[s];

==  Also, there are some grains, and grains are plants.

 reserve p, g for plants;
```

```
P: ex g st grain[g];

== Every animal either likes to eat all plants or all animals much smaller
== than itself that like to eat some plants.

1LIKES: for x holds
                  (for p holds likes[x,p])
               or
                  (for y st smaller[y,x] &
                            (ex p st likes[y,p])
                         holds likes[x,y]);

== Caterpillars and snails are much smaller than birds, which are much smaller
== than foxes, which in turn are much smaller than wolves.

1SMALLER: for x, y st (caterpillar[x] or snail[x]) & bird[y]
                  holds smaller[x,y];
2SMALLER: for x, y st bird[x] & fox[y] holds smaller[x,y];
3SMALLER: for x, y st fox[x] & wolf[y] holds smaller[x,y];

== Wolves do not like to eat foxes or grains, while birds like to eat
== caterpillars but not snails.

2LIKES: for x, y, p st wolf[x] & fox[y] & grain[p]
                   holds not likes[x,y] & not likes[x,p];
3LIKES: for x, y st bird[x]
                 holds (caterpillar[y] implies likes[x,y]) &
                       (snail[y] implies not likes[x,y]);

== Caterpillars and snails like to eat some plants.

4LIKES: for x st caterpillar[x] or snail[x]
               ex p st likes[x,p]

== Therefore there is an animal that likes to eat a grain-eating animal.

begin

  ex x, y, p st likes[x,y] & likes[y,p] & grain[p]

   proof
     consider w, f, b, c, s such that
       WFBCS: wolf[w] & fox[f] & bird[b] & caterpillar[c] & snail[s] by A;
     consider g such that G: grain[g] by P;
     0: not likes[b,s] & (ex p st likes[s,p]) & smaller[s,b]
                        by 3LIKES, 4LIKES, 1SMALLER, WFBCS;
     1: likes[b,g] by 0, 1LIKES;
     2: smaller[f,w] & not likes[w,f] & not likes[w,g]
                        by 3SMALLER, 2LIKES, G, WFBCS;
     3: not likes[f,g] by 2, 1LIKES;
     4: smaller[b,f] by 2SMALLER, WFBCS;
     5: likes[f,b] by 1, 3, 4, 1LIKES;
    thus thesis by G, 1, 5
   end;

== A bit shorter, but messier.

 ex x, y, p st likes[x,y] & likes[y,p] & grain[p]

   proof
     consider w, f, b, c, s such that
       WFBCS: wolf[w] & fox[f] & bird[b] & caterpillar[c] & snail[s] by A;
     consider g such that G: grain[g] by P;
     1: not likes[b,s] & (ex p st likes[s,p]) & smaller[s,b] &
        smaller[f,w] & not likes[w,f] & not likes[w,g] & smaller[b,f]
                        by 1SMALLER, 2SMALLER, 3SMALLER, 2LIKES,
                           3LIKES, 4LIKES, G, WFBCS;
     2: likes[b,g] & not likes[f,g] by 1, 1LIKES;
     3: likes[b,g] & likes[f,b] by 1, 2, 1LIKES;
    thus thesis by G, 3
   end

== Who will make it any shorter?
```

Part V
Mizar MSE Grammar

B Mizar MSE Grammar

B.1 Notation

Name	(in italics) stands for a variable in the grammar, that is a non-terminal symbol.
`token`	(in typewriter font) a token, that is a terminal symbol.
|	separates alternative elements.
( )	groups elements together.
{ }	zero or more occurrences of what is between braces.
{ }$^+$	one or more occurrences of what is between braces.
[ ]	zero or one occurrence of what is between brackets.

B.2 Grammar

Commentaries

Comment ::= `==` *ArbitraryTextToEndOfLine*

Article

Article ::= `environ` *Environment* [`begin` *TextProper*]

Environment ::= {
    Axiom `;` |
    Reservation `;` |
    GlobalConstantsDeclaration `;`
}

TextProper ::= { *Statement* `;` }$^+$

Environment Items

Axiom ::= *LabelId* `:` *Formula*

Reservation ::= `reserve` *Identifier* { `,` *Identiifier* } `for` *SortId*

GlobalConstantsDeclaration ::= `given` *QualifiedObjects*

Text Proper Items

Statement ::= *CompactStatement* |
    DistributedStatement |
    Choice

CompactStatement ::= *Proposition* (*SimpleJustification* | *Proof*)

DistributedStatement ::= [*LabelId* `:`] `now` *Reasoning* `end`

Choice ::= `consider` *QualifiedObjects* [`such that` *Proposition*]
SimpleJustification

Proposition ::= [*LabelId* `:`] *Formula*

SimpleJustification ::= [`by` *Reference* { `,` *Reference* }]

Reference ::= *LabelId*

Reasonings and Proofs

Proof ::= `proof` *Reasoning* `end`

Reasoning ::= { *SkeletonItem* `;` | *Statement* `;` }$^+$

SkeletonItem ::= *Assumption* |
Generalization |
Conclusion

Assumption ::= `assume` *Proposition*

Generalization ::= `let` *ExplicitlyQualifiedObjects*

Conclusion ::= `thus` *CompactStatement*

Formulas

Formula ::= *AndOrFormula* |
ConditionalFormula |
BiconditionalFormula |
QuantifiedFormula

AndOrFormula ::= *ConjunctiveFormula* |
DisjunctiveFormula

DisjunctiveFormula ::= *ConjunctiveFormula* { `or` *ConjunctiveFormula* }

ConjunctiveFormula ::= *UnitFormula* { `&` *UnitFormula* }

UnitFormula ::= { `not` } (*AtomicFormula* | `(` *Formula* `)`)

AtomicFormula ::= *Equality* |
Inequality |
PredicativeFormula |
`contradiction` |
`thesis`

Equality ::= *Term* `=` *Term*

Inequality ::= *Term* `<>` *Term*

PredicativeFormula ::= *PredicateId* [`[` *Term* { `,` *Term* } `]`]

Term ::= *ObjectId*

ConditionalFormula ::= *AndOrFormula* `implies` *AndOrFormula*

BiconditionalFormula ::= *AndOrFormula* `iff` *AndOrFormula*

QuantifiedFormula ::= *UniversalFormula* | *ExistentialFormula*

UniversalFormula ::=
  `for` *QualifiedObjects* [`st` *Formula*] `holds` *Formula* |
  `for` *QualifiedObjects* [`st` *Formula*] *QuantifiedFormula*

ExistentialFormula ::= `ex` *QualifiedObjects* `st` *Formula*

Object Declarations

QualifiedObjects ::= *ImplicitlyQualifiedObjects* |
  ExplicitlyQualifiedObjects |
  ExplicitlyQualifiedObjects `,` *ImplicitlyQualifiedObjects*

ImplicitlyQualifiedObjects ::= *ObjectList*

ExplicitlyQualifiedObjects ::= *QualifiedSegment* { `,` *QualifiedSegement* }

QualifiedSegment ::= *ObjectList* (`be` | `being`) *SortId*

ObjectList ::= *ObjectId* { `,` *ObjectId* }

Identifiers

LabelId ::= *Identifier*

SortId ::= *Identifier*

PredicateId ::= *Identifier*

ObjectId ::= *Identifier*

IdentifierChar ::= `A..Z` | `a..z` | `0..9` | `_`

Identifier ::= { *IdentifierChar* }$^+$ { `'` }

Remarks

- Semicolon followed by `begin`, `end` or end-of-text can be omitted.
- A *QualifiedSegment* in *Generalization* must use `be`; in all other contexts, `being` must be used.
- A *Reasoning* must include at least one *Conclusion* .
- The *Reservation* construct lets us use *ImplicitelyQualifiedObjects* where the *SortId* is not specified explicitly for an *ObjectId* . The *SortId* of such an *ObjectId* is the one that has been reserved for the *ObjectId* in *Reservation* .

- The atomic formula `thesis` can save a lot of writing. `thesis` is always equivalent to the formula that still remains to be proven in order to complete the innermost *Proof* . Thus, any usage of `thesis` outside of a *Proof* is meaningless.

  Consider a *Proof* . Initially, the `thesis` is equivalent to the *Proposition* for which the *Proof* is written. The meaning of `thesis` is affected by each *SkeletonItem* in the way that to many appears mysterious. In a correct *Proof* , the last meaning of `thesis` should be `not contradiction`, that is true.

- The *UniversalFormula* has more variants than in BB version. The construct:

  `for` someQualifiedObjects `st` aFormula1 `holds` aFormula2;

  is equivalent to

  `for` someQualifiedObjects `holds` aFormula1 `implies` aFormula2;

  And also, if the scope of the quantifier is a *QuantifiedFormula* , then `holds` may be omitted.

B.3 Problems with syntax

B.3.1 A question

(syntax01q.mse)

```
== You are given a number of syntactically incorrect formulas.
== Your job is to make them syntactically correct by inserting parentheses
== (no other visible symbols are permitted)
== in such a way that the formulas become not only syntactically correct
== but also are accepted by the Mizar checker (without any errors).
== That is, all formulas you obtain should be tautologies.

environ
        given a being PERSON;

        == If you wish, you may understand the abbreviations as follows:
        == K[a]         a is a knight
        == N[a]         a is a knave
        == S[a]         a is short
        == T[a]         a is tall
        == H[a]         a is handsome
        == You should understand however that the above suggestion is
        == completely immaterial since tautologies are independent of the
        == interpretation.
begin
        == Example. For the syntactically incorrect

        1: K[a] implies T[a] & T[a] implies H[a] & K[a] implies H[a];

        == the solution can be:

        1: ((K[a] implies T[a]) & (T[a] implies H[a]) & K[a]) implies H[a];

        == End of example

 1: K[a] implies S[a] iff S[a] or not K[a];

 2: K[a] implies T[a] implies T[a] implies H[a] implies K[a] implies H[a];

 3: K[a] implies T[a] & T[a] implies H[a] implies K[a] implies H[a];

 4: K[a] implies T[a] implies K[a] implies K[a];

 5: K[a] implies S[a] & N[a] implies T[a] implies K[a] & N[a] implies
                                                                S[a] & T[a];

 6: K[a] implies S[a] & N[a] implies T[a] & K[a] or N[a] implies S[a] or T[a];

 7: K[a] implies S[a] & N[a] implies T[a] & K[a] or N[a] & not S[a] implies
                                                                       T[a];
```

```
8: K[a] implies S[a] & S[a] implies K[a] iff K[a] iff S[a];

9: K[a] & S[a] or not S[a] & not K[a] iff K[a] iff S[a];
```

B.3.2 Its answer

(syntax01a.mse)

```
environ
  given a being PERSON;
begin

1: K[a] implies (S[a] iff S[a] or not K[a]);

2: (K[a] implies T[a]) implies
             ((T[a] implies H[a]) implies (K[a] implies H[a]));

3: ((K[a] implies T[a]) & (T[a] implies H[a]))
             implies (K[a] implies H[a]);

4: (K[a] implies T[a]) implies (K[a] implies K[a]);

5: ((K[a] implies S[a]) & (N[a] implies T[a]))
             implies (K[a] & N[a] implies S[a] & T[a]);

6: ((K[a] implies S[a]) & (N[a] implies T[a]) & (K[a] or N[a]))
             implies S[a] or T[a];

7: ((K[a] implies S[a]) & (N[a] implies T[a]) & (K[a] or N[a])
             & not S[a]) implies T[a];

8: ((K[a] implies S[a]) & (S[a] implies K[a])) iff (K[a] iff S[a]);

9: (K[a] & S[a] or not S[a] & not K[a]) iff (K[a] iff S[a]);
```