INPUT LANGUAGE

The **input language of ExpertPRIZ** consists in effect of two languages: that for specifying **problems** and that for specifying **concepts**. As the syntaxes of the two are identical (with two exceptions), they will be described together. We shall only distinguish them as separate when differences occur. Everything that will be said about models (without having defined them) holds true both for problem models and for concept models, depending which language we have in mind.

1. Notations

1.1. Agreements on notations

The meaning of the notations used in the following text is as follows:

1) Everything printed in **boldface** must be represented exactly in the same way in the corresponding statements of texts in input language, e.g. the keyword **text**;

2) Parts of specifications enclosed between the characters **<**and **>** are the language elements which will be specified later. For example, **<id>** notes any identifier;

3) Parts of statements enclosed between the characters **[** and **]** are not compulsory, i.e. they can be left out. For example, **!file <filename> [#<entry>]** means that the possible statement is either **!file <filename>** or **!file <filename> #<entry>**;

4) Parts of text followed by three dots **...** may be repeated any number of times, e.g. **<id>...** notes the list of identifiers;

5) A vertical bar | between two parts of a specification means that the statement contains one of them. For example, **!clear [y|n]** means that the possible statement may be **!clear y**, **!clear n**, or **!clear**.

1.2. Statements

The input language consists of statements. Every statement occupies a separate line with the length of not more than 255 characters. Statements are of four types:

1) **specifications,**
2) **problem statements,**
3) **commands,**
4) **comments.**

Specifications determine the conditions of a problem, i.e. they specify objects and relations of a problem model.

Problem statements indicate the objects for whose values are to be found.

Commands are used to perform certain actions, e.g. to start solving a new problem, to read input from a file, etc.

Comments are used to add explanatory texts.

1.3. Names, keywords and constants

Identifiers consist of letters and numbers, always beginning with a letter. The underline character _ is regarded equivalent to letters. Identifiers are distinguished by the first eight characters. Capital and lower case letters are regarded different. Identifiers are for example

x
A333
Identifier
obj_name

_
__13

Strings of characters where any of these rules are ignored, are not regarded as identifiers, e.g.

3_cats (because the string begins with a number)
X7$ (because it contains an illegal character $)

The following identifiers are identical, because the first eight characters match:

Long_identifier
Long_ident

The following identifiers are not identical because capital and lower case letters are considered different:

TRIANGLE
triangle

Identifiers are used as the **names** of objects and concepts. Structured objects can be referred to with **compound names**. Compound names may consist of up to nine identifiers, separated from each other by dot .. Thus the name

FIGURE.SQUARE.DIAGONAL
denotes the component DIAGONAL of the component SQUARE of the object FIGURE.

The input language has four **keywords**:

numeric
text
undefined
super

Keywords are distinguished by first three characters; thus they may be replaced by the abbreviations - **num, tex, und, sup**. Keywords cannot be used as identifiers, nor can identifiers beginning with the letters **num**, **tex**, **und** or **sup** be used.

Numeric constants are written in their conventional form:

21
3.14
-0.209
0.00001

Numbers are allowed to be written in exponential form. To denote an exponent, the letter **e** or **E** is used:

1e-5
-1.14E6

Absolute values of numeric constants may not exceed the limits 1e-99 and 1e99.

Any string of characters enclosed into apostrophes is a **textual constant**. Textual constants are for example:

'This is a textual constant'
'33333'
'$ 125.-'

If a text contains an apostrophe, it must be doubled, e.g.:

'It''s a text containing an apostrophe'

Empty string '' as textual constant is allowed.

1.4. Comments

Comments are lines of the input language beginning with *. The system ignores comments, so you can add them for better readability and comprehensibility, e.g.:

*This is a comment.

2. Specification of objects

2.1. Primitive objects

Primitive objects are real numbers and textual objects. Their values are the corresponding numerical and textual constants. **Numerical objects** are specified by the statement:

<id> **numeric**

and **textual objects** by the statement:

<id> **text**

where **<id>** is identifier not declared previously. As a result of these statements, objects of the corresponding type (**numeric** or **text**) are included into the model by the name **<id>**. For example:

```
* Let us specify a numerical object x1
x1  numeric
* and a text by the name LINE
LINE text
```

Specifying numerical objects is not compulsory. If you use previously undeclared identifiers in some statements, then by default the system will consider them **numeric**.

2.2. Objects of undefined type

Objects of **undefined** type are specified by the statement:

<id> **undefined**

where **<id>** is a previously undeclared identifier. The object with the name **<id>** is included into the model, but the type of this object is not determined yet. You can specify its type later binding it with some other object (not **undefined** type) by means of an equation. The undefined object acquires the type of the object to which it is bound. For example:

```
* element can later acquire arbitrary type
element und
```

2.3. Specifications of structural objects

Structural objects are specified by the statement:

<id> (<name> ...)

where **<id>** is a previously undeclared identifier. By the result of this specification the model includes a compound object with the name **<id>**, the components of which are listed in parentheses.

The components of the structure can be of arbitrary type. The names of the components in the list **<name>...** can be the names of objects specified before, or identifiers not used earlier. In the last case the system assigns the type **numeric** to corresponding objects by default. To refer to the components of the structure, both **<name>** and compound name **<id>.<name>** can be used.

Example:

```
* Specify the structure S1 with the numerical components
* a and b, which are specified by default
S1 (a b)
T text
* the only component of the structure S2 is object T of the
* text type, specified on line above
S2 (T)
* S3 contains structural components as well
S3 (S1 S2 T)
* To refer to the object T, the following names can be used:
* T, S2.T, S3.T, S3.S2.T
```

2.4. Specifying objects by means of concepts

Objects can be specified by means of **concepts** using the following statement:

<id> <concept_name> [<binding> ...]

where **<id>** is not specified earlier and **<concept_name>** is the name of a concept from concept base. **<binding>** is an equation in the form:

<component_name> = <value>

or

<component_name> = <name>

where **<component_name>** is the name of a component of the concept **<concept_name>**. **<value>** is a constant, **<name>** is the name of an object, the type of it must correspond to the type of the concept component **<component_name>**. If **<name>** is previously undeclared, it is classified as **numeric** by default.

Having specified an object in the above-given way, the model acquires a compound object by the name **<id>**, the components of which are of the same name and type as in the concept by which it was specified. The relations between the components of the concept and the equations **<binding>** (if the statement contains any) are also included

into the model. For referring to components of an object defined by a concept, compound names must be used: **<id>.<component name>**.

Example:

```
        * To specify an object Car1, we shall use the
        * concept move, which contains components s,t,v
        * - distance, time and speed respectively, and
        * the equation s = v*t
Car1 move
        * the problem model now includes the objects
        * Car1, Car1.s, Car1.t, Car1.v and the equation
        * Car1.s = Car1.v * Car1.t
        * Let us specify the object Car2
Car2 move  v = Car1.v  s = 1000
        * In addition to the objects Car2, Car2.s, Car2.t,
        * Car2.v and the equation Car2.s = Car2.v * Car2.t,
        * the model now includes the equations
        * Car2.v = Car1.v and Car2.s = 1000
```

2.5. Virtual objects

A **virtual object** is an object of any type, if its name looks like

[<vir_id>**]**

where **<vir_id>** is an identifier, in which the characters *, / and ^ are regarded equal to letters.

The difference between virtual and other objects is following: if a compound object contains components that are virtual objects, then the value of the compound object does not contain the values of its virtual components. When finding a value of such an object, it is not compulsory for the system to find values for its virtual components. In every other way virtual and non-virtual objects are alike.

Example:

```
* The concept move1, description of which is
* s = v*t
* [min]*60 = t
* [h] * 60 = [min]
* [km] * 1000 = s
* [km/h] = [km]/[h]
* contains virtual components [min], [h], [km]
* and [km/h].
* Let us specify the object Car
Car move1  v = 30  [min] =25
* Finding value to Car, the system finds values
```

```
* only for Car.[min], Car.s, Car.v and Car.t and
* outputs only the last three.
?Car
```

2.6. Inheritance (super-concept)

A **super**-concept of a problem or a concept is represented by the following statement:

super <concept_name>
where **<concept_name>** is a name of a concept included in knowledge base. In result of this statement, all the objects and relations that the concept **<concept_name>** contains, are included into the model, i.e. the current problem or the concept inherits all the characteristics of the super-concept.

Example:

```
* Specificaton of the concept move1 from previous
* examples, using the concept move, which contains
* the equation s = v*t, as a super-concept, is the
* following:
super move
[min]*60 = t
[h]*60 = [min]
[km]*1000 = s
[km/h] = [km]/[h]
```

NB! You can have more than one super-concept for a concept or a problem, if there are no coincidences of names. This mistake occurs very easily.
For example, if the specification of the concept C1 contains

```
super C0
...
```

and if we try to specify the concept C2 in the form

```
super C0
super C1
...
```

then there will always be conflicts in names.

3. Specification of relations

3.1. Equations

Relations - computational dependencies between objects - can be defined by means of **equations**, which are divided into three classes:

1) arithmetical equations,
2) logical equations and
3) equivalences.

3.1.1. Arithmetical equations

Arithmetical equations are represented in the following form:

<expression> = <expression>

Expressions **<expression>** may contain:

1) positive numerical constants;
2) the names of **numeric** or **undefined** objects, which are considered to be **numeric** by the system;
3) unary minus -;
4) binary operators:
   - **+** addition,
   - **-** subtraction,
   - ***** multiplication,
   - **/** division,
   - **^** exponentation;
5) parentheses **(** and **)**;
6) names of functions:
   - **sin** sine,
   - **cos** cosine,
   - **tan** tangent,
   - **asin** arc sine,
   - **acos** arc cosine,
   - **atan** arc tangent,
   - **sqr** square,
   - **sqrt** square root,
   - **exp** exponent,
   - **ln** natural logarithm,
   - **log** common logarithm,
   - **abs** absolute value,
   - **sign** sign function.

NB! A function name must be immediately followed by left parenthesis **(**.

The order of operations when finding values to the expressions is following:

1) functions,
2) unary minuses,
3) exponentations,
4) multiplications and divisions,
5) additions and subtractions.

The order of operations can be changed with parentheses. Operations, equal by priority, are carried out from left to right.

NB! Exponentations are also carried out from left to right. Therefore the following expressions are not equal:

2^3^2 (= 64) is not equal to **2^(3^2) (= 512)**

The meaning of an arithmetical equation is:

- If, out of objects bound by an equation, only one object is without a value, then it can be found out by solving this equation. Which means that the system consideres the equation computable for all the objects it contains. Whether the equation is actually computable, becomes clear only in the course of computations, i.e. after the problem statement.
- An equation can be entered even if it contains objects which already have values or whose values can be found from other relations. An equation is ignored if it contains no objects at all.

Example:

```
        * Let us bind the objects s, v0, t and a by an
        *  arithmetical equation
s = v0*t + a*t^2/2
        * We can add another, equivalent equation
v0*t + a*sqr(t)/2 = s
        * Which one the system will use, cannot be determined beforehand
```

3.1.2. Logical equations

Logical equations are represented in the following form:

<name> = <extended expression>

where **<name>** is the name of an object of numerical type. If **<name>** is previously undeclared identifier, the system defines it as of numerical type.

Expression **<extended expression>** is similar to arithmetical expressions. In addition, it contains at least one of the following operators or expressions:

1) relational operators
   - **lt** less than,
   - **le** less than or equal to,
   - **gt** greater than,
   - **ge** greater than or equal to,
   - **eq** equal to,
   - **ne** not equal to;

2) logical operators
   - **and**,

or,
not;

3) conditional expression
if then else fi.

Relational operators allow to compare numerical values and expressions, and objects of any type. The result of a relational operation is always numerical. If the condition is satisfied, the result will be 1, otherwise it will be 0.

Logical operators correspond to logical and, logical or, and logical not. Operands must be of numerical type. Zero is considered as logical false and nonzero value as true. These
operators also give a numerical value (0 or 1, i.e. false or true) as a result.

Conditional expressions allow to define values of objects depending on some conditions.

The order of these operations in computing values to the expressions is following:

1) logical not,
2) arithmetical operations,
3) relational operations,
4) logical and,
5) logical or.

The order of operations can be changed using parentheses. Operations, equal by priority, are carried out from left to right.

The meaning of a logical equation is:

- If the value of <extended expression> can be computed, then the value of the object <name> (which stands on left side of the equation) can also be computed. Thus, logical equations are used by the system only in one direction -this is the difference between logical and arithmetical equations.

Example:

```
* cond1 is nonzero if a is greater then the sum of b and c
cond1 = a gt b+c
* cond2 is true if name1 is lexicographically less than name2
name1 text
name2 text
cond2 = name1 lt name2
* n is 1 if x is less than 0;
* n is 2 if x is greater than 0;
* n is 0 if x equals 0:
n = if x lt 0 then 1 else if x gt 0 then 2 else 0 fi fi
```

3.1.3. Equivalences

Equivalences are represented in the following form:

<name1> = <name2>

or <name1> = <value>
where **<name1>** and **<name2>** are names of objects of any type (except **numeric**) and **<value>** is a numerical or textual constant. If **<name2>** is previously undeclared identifier, the system defines it as of numerical type.

Equivalences are used for three purposes:

1) To equalize a textual object with a textual constant or another object of textual type.

Example:

```
Text1 text
Text2 text
* equalize Text1 with a constant
Text1 = 'a textual constant'
* equalize the objects Text1 with  Text2
Text1 = Text2
```

2) To assign type to an undefined object **<name1>**. In this case the right side of the equation is a constant or an object of any type. By including this equation into a model the object **<name1>** acquires the type of the object or the constant from the right side of the equation and becomes equal to this object or constant.

Example:

```
u1 undefined
u2 undefined
u1 = n
* u1 is now equal to n and of the type numeric
Car  move
u2 = Car
* u2 is now identical with the object Car
```

3) To "equalize" compound objects. In case **<name1>** and **<name2>**are compound objects, the corresponding components of these objects are equalized, in fact. Furthermore, **<name1>** and **<name2>** may contain different number of components. The corresponding components are equalized until components of one of the objects are exhausted. The types of respective components of compound objects must be compatible, i.e. each component of the object whose components' number is smaller, must meet the following conditions:

- if this component is of a primitive type, then the type of the corresponding component of the other object must be the same;
- if this component is a compound object, then the corresponding component of the other object must also be a compound object and their types must be compatible.

One equation binding compound objects generates one or more equations that bind primitive components of these objects into the model.

Example:

* Let us specify the objects as follows:

s1 (a b)
s2 (c)
t1 text
t2 text
S1 (d s1 t1)
S2 (e s2 t2 f)

* A nonarithmetical equation

S1 = S2

* generates the following relations into the model:
* d = e
* a = c
* t1 = t2

3.2. Relations given by built-in functions

Besides equations, ExpertPRIZ allows you to represent relations between objects by built-in functions. For this purpose the system has a number of convenient functions - for iterative computations, for using expert systems and database queries, etc. Complete list of available built-in functions and their specifications is given in ..... .

Relations implemented by built-in functions are represented by statements in the following form:

<input_obj>... -> <output_obj> {<function_name>}

where **<input_obj>...** contains the names of the input objects, **<output_obj>** is the name of the output object, and **<function_name>** - the name of the function implementing the relation. If the list of input or output objects contains previously undeclared identifiers, then by default the system defines the objects with these names as of **numeric** type.

The meaning of the relation specified in the above-given way is the following: if the values of input objects **<input_obj>...**are computable, then the value of an output object **<output_obj>**can be computed by means of the function **<function_name>**.

The number of input objects of the relation, their order and types must be legal, i.e. they must meet the requirements of the corresponding function, which implements the relation; **<function_name>** must also be that of an existing function. Fulfillment of these requirements is not checked by the system, when a relation is included in the model, but a failure to meet them results in mistakes in the course of problem solving.

Example:

* There is a function that uses the kernel of expert

```
        * system for problem solving. The name of the expert
        * knowledge base is the input object and the
        * result found by the system is the output object.
base text
base = 'ROAD.EXP'
base -> kf {expert}
```

3.3. Relations with subtasks

Relations implemented by functions can contain subtasks, i.e. problems for finding values to some objects through the values of some other objects by means of the current model. A relation with subtasks is represented in the following form:

<subtask>... <input_obj>... **->** <output_obj> **{**<module_name>**}**

Each subtask **<subtask>** has the form:

(<sub_input_obj>... **->** <sub_output_obj> **)**

where **<sub_input_obj>...** are the names of the input objects of the subtask and **<sub_output_obj>** - the name of the output object of the subtask. If they contain previously undeclared identifiers, the system defines them as of the type **numeric** by default. The rest of the statement is identical to the description of a relation without subtasks (see 3.2.). Like the input and output objects of the relation, the input and output objects of subtasks must be listed in proper order, their number and types must be legal according to the description of the corresponding built-in function **<function_name>**.

The meaning of the relation with subtasks is the following: if all the values of input objects **<input_obj>...** are computable, and all the subtasks **<subtask>...** are solvable, then the value of the output object can be computed by means of the function **<function_name>**.

Example:

```
        * Let us specify a problem for computing the table of
        * squares and cubes of integers from 1 to 10,
        * using the function table
x_square = x^2
x_cube = x^3
first = 1
last = 10
step = 1
        * A line of the table is the following: x  x^2  x^3
line  (x  x_square  x_cube)
result text
```

```
file text
file = 'TABLE.TAB'
        * The following is the relation implemented by the
        * function table
(x -> line)  first last step file -> result {table}
        * Having issued the statement to compute the object result,
        * we'll get the expected table with ten lines and three
        * columns. The table appears in the window and is written
        * into a text file named TABLE.TAB. The object result
        * obtains the value 'End of table'.
? result
```

4. Problem statement

There are two kinds of **problem statements**:

1) to compute **values** of objects,

2) to find **analytical expressions** for dependences of some objects (of **numeric** type) on some other objects of numerical type.

The language for specifying concepts does not contain problem statements. Having received a problem statement, ExpertPRIZ at first verifies whether the problem is solvable on the current model. If the answer is positive, the system carries out corresponding computations by means of the algorithm it found, and outputs the results - the values of the objects or expressions.

4.1. Computing values

The problem for computing values of objects is represented by a statement in the following form:

? [<name>...]

where **<name>...** contains the names of objects of a problem model.

The meaning of this statement is following:

- to compute the values of the objects **<name>...** . If the problem statement does not contain the list of objects, then the system will find values of all the objects (including virtual ones) that are computable without using relations with subtasks.

Example:

```
* Let us specify a problem
Car1 braking [km/h]=100  kf=0.7
```

```
Car2 braking [km/h]=90  kf=Car1.kf
d = Car1.s - Car2.s
* Compute the difference of braking distances d
? d
* We can compute the values of all the objects
?
```

4.2. Finding dependences

The problem for finding analytical dependences of some numerical objects on some other numerical objects is represented by a statement in the following form:

? [<name1>...] **->** [<name2>...]

where **<name1>...** and **<name2>...** are the names of some numerical or compound objects from the problem model. In case of compound objects, all their nonvirtual primitive components must be of **numeric** type.

The meaning of this statement is following:

- to find analytical formulas for the objects **<name2>...** (in case of compound objects, for all their nonvirtual numerical components), which express their dependence on the objects **<name1>...** (if **<name1>** is a compound object, then on its primitive components). If the list **<name1>...**is missing, then the resulting expressions are constants -the same which we get as a result of computing values of objects. If the list **<name2>...** is missing, then expressions are found for all those objects, where it is possible.

Example:

```
* Let us specify a problem
Car1  braking [km/h]=100  kf=kf
Car2  braking [km/h]=90  kf=Car1.kf
d = Car1.s - Car2.s
* Find the dependence of difference of braking
* distances  d  on the friction factor kf
?  kf -> d
```

5. Commands

Command of the input language is a statement beginning with an exclamation mark ("**!**") followed by the command name (in lower-case letters) and parameters, if the command has any:

!<command_name> [<parameter> ...]

The commands are following.

!clear [y|n]
for deleting the current problem model.

Possible forms of that command:
!clear - asks confirmation about clearing;
!clear y - clears the model without any questions;
!clear n - does nothing.

!reset
resets the problem model (deletes the values of all the objects existing in the current model).

!show
displays the contents of the current problem model.

!file <filename> [**#**<entry name>]
reads input from a file. If the name of the file <filename> does not contain extension, the default extension "**.TXT**" is added. One file can contain several parts of input text which can be executed separately. The beginning and the end of an entry in the file are text lines in the form: **#**<name>.

Examples:

!file problem #Car - processes the input text of the entry named Car from the file PROBLEM.TXT. The text is processed from the text line that follows the line "#Car" to the line of the same form, or up to the end of the file (if the end of the entry is not marked).

!file problem - the whole text from the file PROBLEM.TXT
is executed.

!expert <knowledge base name>
starts consultation course with an expert system. The parameter determines the expert knowledge base file name. If the file name does not contain extension, the default extension "**.EXP**" is added.

Example:
!expert demo - starts the demonstration according to the expert knowledge file DEMO.EXP.

!use <concept base name>

opens the concept base. In case the concept base file is not found, makes a new (empty) one with the given name and opens it. If the concept base name does not contain extension, the default extension "**.CPT**" is added.

Example:

!use demo - switches to the concept base DEMO.CPT.

!concept <concept name>

saves the text following this command as a new concept under the given name. The text is considered as a concept description up to the next line beginning with "!", or to the end of the input text. The end mark of the concept description (line beginning with "!") is skipped.

The concept will be saved to the concept base currently open.

For example, the following input lines describe a concept square and add it to the current concept base:

```
!concept square
* Square  a - side; d - diagonal; S - area; p - perimeter.
d^2 = 2*a^2
S = a^2
p = 4*a
!
```