User guide

## TopoL for Windows

# ToPas language

**TopoL Software s.r.o.**

Information in this document is subject to change without any prior notice. While every precaution has been taken in the preparation of this manual, authors and TopoL Software s.r.o. company assume no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.
No part of this manual shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, desktop publishing, recording or otherwise, for any purpose without written permission from TopoL Software s.r.o. company.

© 2000 TopoL Software s.r.o. All rights reserved.

Microsoft and MS-DOS are registered trademarks and Windows is trademark of Microsoft Corporation.
All other product and brand names are trademarks, registered trademarks, or service marks of their respective owners.

# Content

**TOPAS LANGUAGE ..... 5**

**TOPAS TOKENS ..... 6**

**TOPAS TYPES ..... 7**

**EXPRESSION SPECIFICATION ..... 8**

OPERANDS ..... 8
*Constant values ..... 9*
*References to database tables ..... 9*
*References to internal attributes of graphical objects ..... 11*
*Internal functions ..... 16*
*Expressions in parenthesis ..... 21*
*Constant references ..... 22*
*Variable references ..... 22*
OPERATORS ..... 22
*Arithmetical operators ..... 22*
*Logical operators ..... 23*
*Relational operators ..... 24*
*String operators ..... 24*
*Date operators ..... 24*
*Typecasting operators ..... 24*
OPERATOR PRECEDENCE ..... 25
VALUE LIST EXPRESSIONS ..... 26
*Evaluation of list expressions ..... 26*
*Using value list expressions ..... 26*
*Mathematical qualifiers ..... 27*

**DECLARATION PARTS ..... 29**

CONSTANT DECLARATION ..... 29
VARIABLE DECLARATION ..... 29
DECLARATION OF ALIAS NAMES FOR DATABASE TABLES ..... 29
DECLARATION OF ALIAS NAMES FOR BLOCKS OF GRAPHICAL OBJECTS 30

**STATEMENTS ..... 31**

ASSIGNMENT STATEMENT ..... 31
OUTPUT PROCEDURES ..... 31
COMPOUND STATEMENT ..... 31
CONDITIONAL STATEMENT ..... 32
REPETITIVE STATEMENTS ..... 32

**COLOUR AND SYMBOL FORMAT ..... 33**

COLOUR NUMBER FORMAT FOR VECTOR DATA ..... 33
SYMBOL NUMBER FORMAT FOR VECTOR DATA ..... 33

# ToPas language

Utilisation of many analytical tools in TopoL is often dependent on user's chance to define quite complex conditions for their tasks and program ability to process such conditions. It is possible in TopoL to utilise internal programming language ToPas for the definition of conditions in variety of functions. ToPas language syntax is derived from the common programming language Pascal.

TopoL enables an user to use ToPas language version limited to expressions in following tools:

- definition of labels for graphical objects
- definition of queries for selection of graphical objects
- definition of queries for selection of records in database table browser
- definition of expressions for replace of database table item values
- definition of cartodiagrams

Full version of ToPas language can be used for definition of information forms, which is one of application form types.

Language description outlined in following chapters provides detailed list of language features. It cannot be in any case considered as a course of programming.

# ToPas tokens

Tokens are the smallest meaningful units of text in ToPas expressions or programs. They are categorised as special symbols, identifiers, keywords and constant values. Tokens are delimited by any number of separators.

**Special symbols** are characters or their pairs with one or more fixed meanings. ToPas uses following special symbols:

+ - * / ( ) [ ] = , @ \ % ! : . .. < > <= <> >= ; :=

**Identifier** denotes constants, variables, alias names for database tables or blocks of graphical objects, internal functions, graphical object attributes, database table items and join names in database model. Identifier is a character string, which must begin with a letter (character a..z and A..Z) or an underscore (_). Letters, digits, or underscores are allowed after the first character. ToPas language (identifiers, key words) is not case sensitive, so you can use either uppercase of lowercase characters. Some identifiers has special meaning and are considered as keywords.

List of **keywords**:

| | | | | |
|---|---|---|---|---|
| and | begin | block | boolean | const |
| database | datum | div | do | double |
| downto | else | end | for | false |
| if | list | longint | mod | not |
| of | or | repeat | shl | shr |
| string | then | to | true | until |
| var | while | write | writeln | xor |

**Constant values** are categorised as integers, real numbers, strings, logical values and date constants.

**Separators** are spaces, line ends and comments.

**Comment** is any text closed in braces ({}).

Program lines have a maximum length of 255 characters.

# ToPas types

There are five predefined simple types which are analogous to types of database table items. Then there are five predefined structured types for processing of value lists. In some cases ToPas enables us to use another two special types.

There are following predefined simple types:

- integers (whole numbers, numeric values without decimals) - type name longint
- real numbers (numeric values with decimals or in exponential form) - type name double
- strings (character string values) - type name string
- logical values (yes/no) - type name boolean
- date - type name datum

The structured types are lists of simple type values. Type names are in form of list of simple_type. These lists contain components of type simple_type, which are indexed in ascending order starting from value of 1. Maximum number of components in list is 16380. There are following predefined structured types:

- list of integer values - type name list of longint
- list of real number values - type name list of double
- list of string values - type name list of string
- list of logical values - type name list of boolean
- list of date values - type name list of datum

There are following special types in ToPas language:

- alias name for a database table
- alias name for a block of graphical objects

These types are introduced for special purposes and limited in their use. They can identify references to database table items or internal attributes of graphical objects. They can be also used for parameters in special functions associated with aliased object types.

# Expression specification

Expression syntax chosen during ToPas design enables us to utilise not only common mathematical, relational and string operators and functions, but also to access internal attributes of graphical objects and information stored both in primary (internal) and secondary (external) database tables attached to graphical objects. There are also implemented statistical operators and general mathematical qualifiers to process relations one-to-many (1:N) in database models.

Expressions in ToPas language are made up from operands and operators according to strictly defined syntactical rules. During expression evaluation, operands and operators are calculated using precedence rules from left to right.

ToPas is able to process references to secondary (external) database tables including support for relation one-to-many (1:N). These references can represent several values (list) of the same type. In that case lists are used during calculation and expressions are automatically evaluated for all possible results of such references. That is why structured types and constructs for value list processing are included in ToPas. For more precise specification of their processing, it is possible to use general mathematical qualifiers “exists“ and “for all“ in conjunction which relational and logical operators.

While specifying expressions, it is necessary to take into account data types of references to database tables and internal attributes of graphical objects, because it is not possible to combine arbitrarily operands and expressions of different types with operators and expressions resulting in value lists. For example, it is not possible to multiply integer value by string, even though this string forms a number. It is also not possible to subtract two operands representing value lists, because there is no simple method how to generally define evaluation of such an expression. In that case, incompatibility of operands each other and/or with operator can be sometimes solved through typecasting operators. Integer values can be always used instead of real number values. It is also possible to forbid some database table join evaluation, if it represents more than one value (result). Operands representing value lists can be also limited to single value through use of some functions (e.g. statistical) or indexes.

To simplify the access to database tables and graphical objects, references are always related to current graphical object and/or current database table record.

- In definition of labels for graphical objects, queries for selection of graphical objects, carto-diagrams, and information forms, such references are related to particular block, object type (point, polyline, area), object group and incidentally also to attached primary database table and its database model. Then expressions are evaluated for particular graphical object.
- In definition of queries for selection of records in database table browser and expressions for replace of database table item values, such references are related directly to particular database table and its database model. If this table is also primary one, expressions can also reference graphical objects from particular block, object type (point, polyline, area) and object group closely linked with this table. Then expressions are evaluated for particular database table record.

## Operands

ToPas version limited to expressions includes following operand categories:

- constant values
- references to database tables
- references to internal attributes of graphical objects
- internal functions
- expressions within parenthesis

Full version of ToPas language includes additional operand categories:

- constant references
- variable references

## Constant values

ToPas language enables us to specify constant values of all predefined simple types, i.e. integer, real number, string, logical, and date values.

### Integer values

Values are sequences of digits 0-9 (without spaces). Negative values are denoted by minus sign (-) preceding the first digit. Values out of range -2,147,483,647 to 2,147,493,647 is internally processed as real numbers.

It is also possible to specify integer values in hexadecimal form. In that case, values are sequences of digits and characters a-f, A-F preceding with the dollar symbol ($).

**Example:** *2, -100, 20000000000*
*$1F, $FFFFFFFF, -$FF*

### Real number values

Values are composed from integer and decimal part separated by decimal point (no comma!) and/or integer exponent. Exponent is preceded with character E (or e) and incidentally with sign (+,-)

**Example:** *1.0, -529.086, 0.05, 5.5E-6*
*-3.4E+12, 5e6*

### String values

Values are arbitrary character sequences enclosed by apostrophes (') and written on one line. A character string with nothing between the apostrophes ('') is a null (empty) string. Two sequential apostrophes within a character sequence denote a single character, an apostrophe. Then a character string composed from four apostrophes denotes just a single apostrophe.

**Example:** *'This is a string',*
*'An apostrophe '' within string'*

### Logical values

Logical constants are denoted with TRUE and FALSE keywords. Constant value specification is not case sensitive.

**Example:** *True, false*

### Date values

Date values are specified similarly to their presentation in database record editors within TopoL They are in the form of day.month.year. Year indication can be abbreviated, so that 94 and 1994 are considered as equal. Date value constant need not contain any spaces and must be valid calendar date. Leap years are properly calculated during expression evaluation, but the calendar reform in 1582 is not taken into account. So all date constructs behave correctly in view of Gregorian calendar.

**Example:** *24.7.1963, 31.12.94*

## References to database tables

These references enables us to access individual record items in database table or table joined through database model. References to database tables have to start with the symbol @. Database item name specification is also not case sensitive. Types for values resulting from these references are deduced from types of database items.

To simplify the access to database tables, references are always related to current database table and its record.

- In definition of labels for graphical objects, queries for selection of graphical objects, carto-diagrams, and information forms, current database table and model for a particular expres-

sion (program) are primary database table and its model attached to particular block, object type and group, for which this expression (program) is calculated.

- In definition of queries for selection of records in database table browser and expressions for replace of database table item values, current database table and model for a particular expression (program) are currently processed table and its model.

Reference to an item of current database table has following form:

@ item_name

**Example:** *@NAME, @REGION*

If we want to reference items in tables attached through database models, it is necessary to specify full identification of the table, where the referenced item is located. This identification is composed from references to database joins, which form the full path through the database model to reach the referenced table. References to database joins has to be separated by the dot symbol (.). Database joins can be specified in two manners: join number or join name (symbolic identifier). Both parameters are indicated in database model description files, where each join must be assigned with its number. Join name can be defined through the first word in INFO keyword value.

Reference to an item in a table attached through a database model has following form:

@ join1. ...joinN.item_name

**Example:** *@1.BUDGET,*
*@2.1.PLACE,*
*@Pollution.Station.NO*

If there exist no record in database table attached to particular object/record, the calculation is interrupted and the result for this expression is not evaluated.

Above given methods for specification of references to database tables are sufficient for the most of analytical tasks performed in TopoL. In case of need for detailed access to items of arbitrary database tables, ToPas includes methods for specification of extended references to database tables.

**Extended references to database tables**

Above given methods for the specification of references to database tables are sufficient for the most of analytical tasks performed in TopoL. In case of need for detailed access to items of arbitrary database tables, ToPas includes features for specification of complex references. Specification of such references is based on extended syntactical constructs for the access to database tables.

It is possible to address a particular database table record using indexes. **Index** is an integer expression in range 1..record_number, which defines the position of record within database table. It is closed in square brackets ([]) and placed just after symbol denoting reference to database table (i.e. @). For the current database, index meaning slightly differs according to performed tasks:

- In definition of labels for graphical objects, queries for selection of graphical objects, carto-diagrams, and information forms, indexes correspond to the number of graphical object and then address records attached to these particular objects.
- In definition of queries for selection of records in database table browser and expressions for replace of database table item values, indexes correspond directly to the position of physical record.

In references to tables attached through database models, it is possible to place the index just after symbol denoting a database join (i.e. .). In that case, indexes correspond to the position in the record set attached to particular record. They can range from 1 to Num(reference) - for detailed meaning and description of this feature see section Value list expressions.

**Example:** *@[10]REGION,*
*@1.[NMax(@1.BUDGET)]MONTH*

While ToPas version limited to expressions supports references to the current database only, it is possible in full ToPas version to reference other database tables, which **alias name** is defined in database alias declaration part. In reference specification, this alias precedes @ operator. It is also necessary (with exception of the current database) to specify addressed record using index, which in such a case corresponds directly to the position of physical record.

The reference to a particular database table has following form:
`database_id table_reference`

**Example:**
*Budgets@[10]AMOUNT*
*Current@1.BUDGET*

For easier manipulation with references to a particular database table, ToPas includes following functions, which enable us to find number of records and their validity:

MaxRecord (database)

returns number of all records in table database as an integer value

ValidRecord (database, index)

returns a logical value indicating, if the record index in table database is valid (not deleted)

**Note:** Database tables can contain records, which are marked deleted. ToPas consider references to deleted records to be incorrect and interrupts the calculation of such an expression. Deleted records can be only accessed, if it is used an index corresponding directly to the position of physical record. In that case, it is possible to take advantage of ValidRecord function, which is appointed to trap deleted records in database tables.

## References to internal attributes of graphical objects

These references enable us to access individual attributes of TopoL graphical objects. References to internal attributes of graphical objects have to start with the percent symbol (%). Geometric features are expressed in the current coordinate system/projection (Gauss-Krüger, S-JTSK) with basic measurement unit of 1 m. Attribute name specification is also not case sensitive.

To simplify the access to graphical objects, references are always related to current block of graphical objects, their type (point, polyline, area) and group and also during evaluation to the current object.

- In definition of labels for graphical objects, queries for selection of graphical objects, carto-diagrams, and information forms, the particular block, type, group and object, for which this expression (program) is calculated, are considered to be the current one.
- In definition of queries for selection of records in database table browser and expressions for replace of database table item values, expressions can also reference graphical objects, if the current table is also primary one. In that case, particular block, object type and group closely linked to this table are considered to be the current one. The graphical object attached to the database record currently evaluated is considered to be the current one.

Reference to an internal attribute of current block has usually following form:

% internal_attribute

References can also have form of function with arguments:

% internal_attribute( v1, ..., vN )

**Example:** *%BinColor, %Area, %XLin(1)*

Above given methods for specification of references to internal attributes of graphical objects are sufficient for the most of analytical tasks performed in TopoL. In case of need, ToPas includes methods for specification of extended references to internal attributes of graphical objects.

**Extended references to internal attributes of graphical objects**

Above given methods for specification of references to internal attributes of graphical objects are sufficient for the most of analytical tasks performed in TopoL. In case of need for detailed analyses, ToPas includes features for specification of complex references. Specification of such references is based on extended syntactical constructs for the access to internal attributes of graphical objects.

It is possible to address a particular graphical object using indexes. **Index** specifies object type and number in a block. It is composed from object type specifier (B, L, P) and an integer expression in range 1..MaxPnt (or 1..MaxLin or 1..MaxAre), which defines the number of graphical object. Index is closed in square brackets ([]), components are separated by comma symbol (,) and placed just after symbol denoting reference to internal attribute (i.e. %).

References to a particular graphical objects have following forms:

% [type,number]internal_atribute

% [type,number]internal_attribute(v1,...,vN)

**Example:** *%[B,20]BinColor, %[P,20]Area*

While ToPas version limited to expressions supports references to the current block only, it is possible in full ToPas version to reference other blocks, which **alias name** is defined in block alias declaration part. In reference specification, this alias precedes % operator. It is also necessary (with exception of the current block) to specify addressed graphical object using index.

The reference to a particular block of graphical objects has following form:

block_id `reference_to_attribute`

**Example:** *Towns%[P,20]Area*

For easier manipulation with references to a particular block, ToPas includes following block attributes:

%MaxPnt

returns maximum number of point in the block as an integer value (i.e. number of valid and deleted points with the exception of point 0). If there are no points in the block, attribute returns value of 0.

%MaxLin

returns maximum number of line in the block as an integer value (i.e. number of valid and deleted lines with the exception of line 0). If there are no lines in the block, attribute returns value of 0.

%MaxAre

returns maximum number of area in the block as an integer value (i.e. number of valid and deleted areas with the exception of area 0). If there are no areas in the block, attribute returns value of 0.

**Note:** TopoL numbers graphical objects (points, polylines, areas) within a block in the ascending order from 0. Each block contains at least object 0, which is always (!) deleted. TopoL does not process this object at all and its attributes are not used. Numbers for valid objects start from 1. During object delete operation, TopoL marks this object as deleted, but does not remove it. Its number is not (!) reused during new object creation. Deleted objects can be removed in Pack block operation only.

Previous note states that blocks can contain objects marked deleted. ToPas does not (!) consider reference to deleted object to be incorrect. Deleted records can be only accessed using indexes. In that case, it is possible to take advantage of Valid attribute, which is appointed to trap deleted objects in block. Other references to internal attributes are undefined for deleted objects and can harm performed calculations.

**Basic identification of graphical object**

Number

returns the number of the graphical object as an integer value

Group

returns the group of the graphical object as an integer value

Class

returns the group name for the graphical object as a string

ObjDate

returns the creation date of the graphical object as a date value

**Basic characteristics of graphical object**

Valid

returns TRUE for a valid object and FALSE for an invalid (deleted) one

Attrib

returns status flags for the graphical object as an integer value

Selected

returns if the graphical object is selected or not selected as a logical value

SelectObj

selects the graphical object and returns a logical value, if it is selected or not selected

DeselectObj

deselects the graphical object and returns a logical value, if it is selected or not selected

---

**Note:**

During object delete operation, TopoL marks the object as deleted, but does not remove it. Its number is not (!) reused during new object creation. So that, blocks can contain objects marked deleted. ToPas does not (!) consider reference to deleted object to be incorrect. Deleted records can be only accessed using indexes. In that case, it is possible to take advantage of Valid attribute, which is appointed to trap deleted objects in block. Other references to internal attributes are undefined for deleted objects and can harm performed calculations.

Reference to Attrib attribute is included for test and development purposes only and its detailed description is not publicly available. Valid attribute result is derived from its value.

---

**Display attributes of graphical objects:**

`BinColor`

returns the colour of the graphical object stored in data (in TopoL internal format) as an integer value

`BinSymbol`

returns the symbol of the graphical object stored in data as an integer value

`ActColor`

returns the currently selected display colour of the graphical object in TopoL (in RGB model) as an integer value

`ActSymbol`

returns the currently selected display symbol of the graphical object in TopoL as an integer value

`SetBinColor(colour)`

changes the colour of the graphical object stored in data (in TopoL internal format) into an integer value of the expression colour and returns its new colour

`SetBinSymbol(symbol)`

changes the symbol of the graphical object stored in data into an integer value of the expression symbol and returns its new symbol

---

**Note:**

Colour of graphical objects stored in data (SetBinColor function argument or BinColor and SetBinColor results) is expressed in the internal TopoL format (see section Colour number format for vector data).

Actual colour of graphical objects used during object painting (ActColor result) is expressed in the standard colour format used in the MS Windows environment. Windows utilise the RGB colour model, which is well-suited particularly for computer monitors.

Functions for colour manipulations are able to translate between both colour representations and control colour field intensity values of RGB model.

Symbol of the graphical object (SetBinSymbol function argument or BinSymbol, ActSymbol, and SetBinSymbol results) is expressed in the internal TopoL format (see section Symbol number format for vector data).

**Geometric outline of graphical object**

`MinX`

returns minimum X coordinate of the geometric outline (bounding box) for the graphical object as a real number value

`MinY`

returns minimum Y coordinate of the geometric outline (bounding box) for the graphical object as a real number value

`MaxX`

returns maximum X coordinate of the geometric outline (bounding box) for the graphical object as a real number value

`MaxY`

returns maximum Y coordinate of the geometric outline (bounding box) for the graphical object as a real number value

**Note:** Geometric features are expressed in the current coordinate system/projection (Gauss-Krüger, S-JTSK) with basic measurement unit of 1 m.

**Basic point geometric features**

`X`

returns X coordinate of the point location as a real number value

`Y`

returns Y coordinate of the point location as a real number value

`Z`

returns Z coordinate of the point location as an real number value (in system variants with the support for 3D vector data)

`Angle`

returns the point symbol rotation angle in radians as a real number value

**Note:** Geometric features are expressed in the current coordinate system/projection (Gauss-Krüger, S-JTSK) with basic measurement unit of 1 m.

**Basic polyline geometric features**

`Length`

returns the polyline length as a real number value

`NumPoints`

returns the number of vertices (polyline points) as an integer value

`XLin(vertex_number)`

returns X coordinate of the vertex `vertex_number` (in range 1..NumPoints) as a real number value

`YLin(vertex_number)`

returns Y coordinate of the vertex `vertex_number` (in range 1..NumPoints) as a real number value

`ZLin(vertex_number)`

returns Z coordinate of the vertex `vertex_number` (in range 1..NumPoints) as a real number value (in system variants with the support for 3D vector data)

**Note:**

Polyline denotes in TopoL a series of lines, connecting each of the NumPoints vertices (polyline points). General internal attributes (e.g. colour and symbol) applies to all lines in polyline.

Geometric features are expressed in the current coordinate system/projection (Gauss-Krüger, S-JTSK) with basic measurement unit of 1 m.

**Additional polyline geometric features**

`XRef`

returns X coordinate of polyline reference point (label point) as a real number value

`YRef`

returns Y coordinate of polyline reference point (label point) as a real number value

`LArea`

returns number of the first area adjoining polyline as a real number value

`RArea`

returns number of the second area adjoining polyline as a real number value

**Note:**

LArea and RArea attributes are only defined for polylines on area boundaries. If topological relations are correctly built, these polylines has some special features. Polylines on area boundary adjoins up to two areas and other polylines can only touch them in nodes (i.e. in their first and last points). If polyline is not on boundary of two areas, one or both attributes LArea and RArea return the value of -1.

Geometric features are expressed in the current coordinate system/projection (Gauss-Krüger, S-JTSK) with basic measurement unit of 1 m.

TopoL numbers valid graphical objects (points, polylines, areas) within a block in the ascending order from 1.

**Basic area geometric features**

Area

returns the area size in $m^2$ or ha (based on TopoL menu switch Area [HA]) as a real number value

NumLines

returns the number of polylines (creating the full boundary of the area) as an integer value

AreLin(polyline_index)

returns the polyline object number for the polyline polyline_index on the full boundary of the area as an integer value

**Note:**

Area denotes in TopoL a series of NumLines connected polylines. If topological relations are correctly built, these polylines has some special features. Polylines on area boundary adjoins up to two areas (see LArea and RArea polyline attributes) and other polylines can only touch them in nodes (i.e. in their first and last points).

TopoL numbers valid graphical objects (points, polylines, areas) within a block in the ascending order from 1.

**Additional area geometric features**

XRef

returns X coordinate of area reference point (label point) as a real number value

YRef

returns Y coordinate of area reference point (label point) as a real number value

Parent

returns the number of area, which surrounds the referenced area (i.e. the referenced area is enclave of another one) as a real number value

IncludesArea

returns TRUE, if the referenced area surrounds another one (i.e. the referenced area includes enclaves), otherwise FALSE

**Note:**

If referenced area is contained in another area (i.e. is its enclave), Parent attribute returns the number of smallest area, which surrounds directly the enclave. If the referenced area is not enclave, Parent attribute returns value of -1.

Geometric features are expressed in the current coordinate system/projection (Gauss-Krüger, S-JTSK) with basic measurement unit of 1 m.

TopoL numbers valid graphical objects (points, polylines, areas) within a block in the ascending order from 1.

## Internal functions

ToPas includes many internal functions for each data type processing:

- mathematical functions
- string manipulation functions
- date manipulation functions
- colour manipulation functions
- statistical functions
- list selection functions
- list sort functions

**Mathematical functions**

Support for mathematical functions in ToPas includes:

Abs(expression)

returns the absolute value of the expression

Odd(expression)

returns TRUE for an odd result of expression and FALSE for an even one

Round(expression)

returns value of the expression rounded to an integer value

Trunc(expression)

returns the value of the expression truncated (rounded toward zero) to an integer value

Int(expression)

returns the integer part of the value of the expression

Frac(expression)

returns the fractional part of the value of the expression

Sqr(expression)

returns the square of the value of the expression

Sqrt(expression)

returns the square root of the value of the expression

Exp(expression)

returns the exponential of the value of the expression, i.e. the base of the natural logarithms raised to the power of the expression

Ln(expression)

returns the natural logarithm of the value of the expression

Pi

returns the value of Pi (i.e. 3.1415926535897932385)

Sin(expression)

returns the sine of the value of the expression (expressed in radians)

Cos(expression)

returns the cosine of the value of the expression (expressed in radians)

Tan(expression)

returns the tangent of the value of the expression (expressed in radians)

ArcTan(expression)

returns the arctangent of the value of the expression (expressed in radians)

The Odd function argument must be an integer. The other function arguments can be integers or real numbers. There are some limitations on argument values for several functions. In case, the argument value does not comply with the limitation, the calculation is interrupted and the result for this expression is not evaluated. The Sqrt function argument must be nonnegative (>=0), the Ln function argument must be positive (>0), the Tan function argument cannot be equal to Pi/2 + k*Pi.

Function Odd returns logical values. Functions Trunc and Round return integer values. Functions Abs a Sqr return results of the same type as their arguments. The other functions returns real number values.

**Note**: An expression representing list of values is possible to use for the argument of all listed functions. In that case, functions return list of corresponding results.

**String manipulation functions**

Support for string manipulation functions in ToPas includes:

Char(expression, index)

returns a string containing a single character located in the string expression on the position index

Ord(expression, index)

returns an integer code of a single character located in the string expression on the position index

Chr(expression)

returns a string containing a single character, to which is converted the integer value of the expression

StrLength(expression)

returns an integer length of the string expression

StrUpCase(expression)

returns a string expression converted to uppercase

StrLowCase(expression)

returns a string expression converted to lowercase

StrTrim(expression)

returns a string expression without leading and trailing spaces

StrInTrim(expression)

returns a string expression without leading and trailing spaces as well as without duplicate spaces inside the string

Language

returns an integer code of the current character code page (character encoding for the current block) - typical results: 1252 (ANSI Windows), 1251 (ANSI Windows Cyr), 1250 (ANSI Windows CE), 437 (ASCII), 850 (PC Latin 1), 852 (PC Latin2), 867 (Kamenických encoding)

StrConvToAsc(expression)

returns the string expression converted to ASCII code (diacritics removal)

StrConvToKam(expression)

returns the string expression converted from the current code page into Kamenických encoding

StrConvToLat2(expression)

returns the string expression converted from the current code page into PC Latin 2 encoding

PrefStr(prefix, expression)

returns TRUE, if the string prefix is found at the beginning of the string expression, otherwise FALSE

SubStr(substring, expression)

returns TRUE, if the string substring is found within the string expression, otherwise FALSE

StrPos(substring, expression)

returns an integer value that is the index of the first character of the first occurrence of the string substring within the string expression, if not found returns value of 0

Like(v1, v2)

returns TRUE, if the strings v1, v2 are similar (are different only in letter case, redundant spaces and diacritics), otherwise FALSE

StrCopy(expression, index, count)

returns a string containing count characters copied from the string expression starting at the position index

StrDelete(expression, index, count)

returns a string after the removal of count characters starting at the position index from the string expression

StrInsert(source, expression, index)

returns a string after insert of source string into the string expression at the position index

ToPas strings can be considered as character arrays. Positions of individual characters are expressed by integer values in range 1..Length. Maximum length of each string is 255 characters. The argument and result types are clearly identified in function descriptions. The current code page used within conversion functions is the code page of the current block.

**Note**: An expression representing list of values is possible to use for the argument expression in all listed functions. In that case, functions return list of corresponding results.

**Date manipulation functions**

Support for date manipulation functions in ToPas includes:

Date(day, month, year)

returns a date value specified as the day, month, year integer arguments

Day(expression)

returns the day of the date expression as an integer value

Month(expression)

returns the month of the date expression as an integer value

Year(expression)

returns the year of the date expression as an integer value

Today

returns the current date (based on computer registers)

Weekday(expression)

returns the weekday of the date expression as an integer value in range 0 to 6 (0 - Sunday, ..., 6 - Saturday)

Leap years are properly calculated during expression evaluation, but the calendar reform in 1582 is not taken into account. So all date constructs behave correctly in view of Gregorian calendar. Operands must represent correct date, otherwise the calculation is interrupted and the result for this expression is not evaluated.

**Note**: An expression representing list of values is possible to use for the argument expression in all listed functions. In that case, functions return list of corresponding results.

**Colour manipulation functions**

Support for colour manipulation functions in ToPas includes:

RedColor(RGBcolour)

returns an integer value, which specifies the intensity of the red colour field extracted from the integer expression RGBcolour

GreenColor(RGBcolour)

returns an integer value, which specifies the intensity of the green colour field extracted from the integer expression RGBcolour

BlueColor(RGBcolour)

returns an integer value, which specifies the intensity of the blue colour field extracted from the integer expression RGBcolour

RGBColor(red, green, blue)

returns an integer value expressing colour in the RGB colour model specified as the red, green, blue integer colour field arguments

WinColor(TopoLcolour)

returns an integer value resulting from the conversion of the colour in the TopoL internal format into the RGB colour model

TopoLColor(RGBcolour)

returns an integer value resulting from the conversion of the colour in the RGB colour model into the TopoL internal format

TopoL for Windows uses two different colour representations. Colour of graphical objects stored in data is expressed in the internal TopoL format (see section Colour number format for vector data). Actual colour used during object painting is expressed in the standard colour format used in the MS Windows environment. Windows utilise the RGB colour model (for 24 bits), which is well-suited particularly for computer monitors. WinColor and TopoLColor functions convert between both colour representations. RedColor, GreenColor and BlueColor functions extracts intensity values (in range 0..255) of individual colour fields (red, green, and blue), which form colours in the RGB model through additive composition. RGBColor function is possible to use for the specification of colours in the RGB model based on their colour field components.

**Statistical functions**

References to secondary (external) database tables, in fact, represent several values (list) of the same type. So that, lists are used during the evaluation of these references and the calculation is automatically performed for all possible results of such references. That is why structured types and other constructs for processing expressions representing value lists are included in ToPas.

Support for value list processing in ToPas includes following statistical functions:

Num(expression)

returns the number of values in the list expression

Result(expression, idx)

returns a value on the position index in the list expression

Max(expression)

returns the maximum value in the list expression

NMax(expression)

returns a position (index) of the maximum value in the list expression

Min(expression)

returns the minimum value in the list expression

NMin(expression)

returns a position (index) of the minimum value in the list expression

Sum(expression)

returns the sum of values in the list expression

Ave(expression)

returns the average of values in the list expression

The arguments of Sum and Ave functions must be lists of numeric values (either integers or real numbers). The other function arguments can be lists of any type. Integer values in range 1..Num(expression) denotes positions (indexes) in value lists.

If the argument of statistical function represents empty list (e.g. there is no record in database table attached to particular object/record), the calculation is (with the exception of Num function) interrupted and the result for this expression is not evaluated. Num function can return a result for the empty list argument. In that case the calculation is not interrupted, because the function returns value of 0.

**Example:**
*Table of month budgets for each shop is linked through join no. 1. To find the maximum month budget for each shop, following expression can be used*

*Max(@1.BUDGET)*

**List selection functions**

These functions provide simplified version of SELECT command, which is well-known from the standard database query language SQL. Function Select can be used to select those values from a list which satisfy a certain condition. Function Sqlsel can be used in a similar way to select values from a certain database table.

List selection functions have following form:

Select (expression, condition)

Sqlsel (database, expression, condition)

Function Select evaluates both function arguments for each value from the list expression. If the argument condition returns TRUE, the corresponding result of expression is added to the resulting list. The argument expression can be a list of any type, while the argument condition must be list or value of logical type. The type of function result is the same as the type of argument expression.

Function Sqlsel evaluates both function arguments for each valid record in database table. If the argument condition returns TRUE, the corresponding result of expression is added to the resulting list. The argument expression can be of any type, while the argument condition must be list or value of logical type. Function result is a list of the same type as the argument expression. It is possible to use an alias name declared for database table within alias name declaration part (in full version of ToPas) or identifier CURRENT predefined for the current database (in any expression).

**Example:**
*Table of month budgets (items BUDGET, MONTH) for each shop is linked through join no. 1. Expression @1.BUDGET determines 12 values for each month linked to a particular object. Condition @1.MONTH = 'JANUARY' can be used to find a corresponding value for January. To find the month budget for January, following expression can be used Select(@1.BUDGET, @1.MONTH = 'JANUARY')*
*To find the maximum value of a certain item (e.g. POPULATION) in the current database, we can use following tip. All values of the item POPULATION is selected using Sqlsel function with always valid condition. The selection result is then used for the Max function argument. The expression has following form:*
*Max(Sqlsel(Current, @POPULATION, TRUE))*

### List sort functions

List sort functions can be mainly used for the creation of sorted tables and other outputs for information forms. Functions can be applied to any list resulted from expression evaluation. Sorting is always performed in the ascending order (from minimum to maximum value).

List sort functions have following form:

Sort (expression, sorder)

Getorder (sorder)

Getindex (sorder, sidx)

Order (expression, sorder, sidx)

Both arguments of Sort function can be lists of any type. These types can be different. Function returns the values from the list expression sorted by values in the sorting list sorder. The type of function result is the same as the type of argument expression.

Function Getorder returns the list of integer values (1..Num(sorder)), which specify the order of values in the sorting list sorder.

Function Getindex returns an integer value (in range 1..Num(sorder)) specifying component position (index) in the sorting list sorder, which has the order position sidx (in range 1..Num(sorder)).

Function Order returns a value from the list expression sorted by values in the sorting list sorder, which is located on the order position sidx.

**Example:**
*Table of month budgets (items BUDGET, MONTH) for each shop is linked through join no. 1. To sort month names by budget in ascending order, following expression can be used*
*Sort(@1.MONTH,@1.BUDGET)*
*It is possible to find month with the second lowest budget with following expression*
*Order(@1.MONTH,@1.BUDGET,2)*

Following three statements are always valid for Sort, Getorder, Getindex and Order function results:

Order(expression, sorder, sidx)

= Result(Sort(expression, sorder), sidx))

Order(expression, sorder, sidx)

= Result(expression, Getindex(sorder, sidx))

sidx = Result(Getorder(sorder),

Getindex(sorder, sidx))

## Expressions in parenthesis

Any expression can be closed in parenthesis and combined with other operands and operators. Expression in parenthesis are treated as a single operand and so that evaluated prior to operators.

Syntax to close an expression in parenthesis:

( expression )

**Example:**

```
(100000 < @1.POPULATION)
 and (@1.POPULATION <= 150000)
```

## Constant references

Full version of ToPas makes possible to include references to constants into expressions. Constants are defined in constant declaration parts. The constant reference has the form of the constant identifier.

**Example:**

```
const MaxCount = 30;
...
begin
  for Count := 1 to MaxCount
  do begin
  ...
  end;
end
```

## Variable references

Full version of ToPas makes possible to include references to variables into expression. Variables are defined in variable declaration parts. The variable reference has the form of the variable identifier.

A specific list component can be accessed in variables of structured types (lists). In that case, the variable identifier is followed by an index which is closed in square brackets ([]). **Index** defines position in the list and can be specified as an expression, which can range from 1 to Num(variable).

**Example:**

```
var Budgets : list of Longint;
    Count : Longint;
begin
  Budgets := @1.BUDGET;
  write ('Budgets:')
  for Count := 1 to Num(Budgets)
  do begin
    writeln(Budgets[Count]:C:10);
  end;
end
```

## Operators

ToPas supports following operator categories:

- arithmetic
- logical
- relational
- string
- date
- typecasting

## Arithmetical operators

Arithmetic operators are used to combine numeric values. Results of their evaluation are also numeric values.

Following operators can combine both integer and real number values. An integer value is their result only in such a case, that operands are integer values and operation is not division.

+ op

op sign identity (unary plus)

- op

op sign negation (unary minus)

op1 + op2

addition of values op1 and op2

op1 - op2

subtraction of value op2 from op1

op1 * op2

multiplication of value op1 by op2

op1 / op2

division of value op1 by value op2

Following operators can be used with integer values only. An integer value is also their result.

op1 div op2

integer division op1 by value op2

op1 mod op2

remainder after integer division op1 by value op2

op1 shl op2

shift the value of op1 by op2 bits to the left (i.e. multiplication by value 2^op2, op2 >= 0)

op1 shr op2

shift the value of op1 by op2 bits to the right (i.e. division by value 2^op2, op2 >= 0)

not op

bitwise negation op

op1 and op2

bitwise and of values op1 and op2

op1 or op2

bitwise or of values op1 and op2

op1 xor op2

bitwise xor (exclusive or) of values op1 and op2

There are some limitations on operand values for several operators. In case, the argument value does not comply with the limitation, the calculation is interrupted and the result for this expression is not evaluated. The operand op2 cannot be equal to 0 for division, integer division and remainder operators. The operand op2 must be nonnegative (>=0) for shift operators.

## Logical operators

Logical operators are used to combine logical values or their lists. Method for evaluation of operator applied to value list(s) is possible to specify using general mathematical qualifiers “exists“ and “for all“. Results of logical operator evaluations are also logical values or their lists.

ToPas includes four common operators:

not op

op negation - result is TRUE, if op is FALSE and vice versa

op1 and op2

logical and of values op1 and op2 - result is TRUE only in case, that both op1 and op2 are TRUE

op1 or op2

logical or of values op1 and op2 - result is FALSE only in case, that both op1 and op2 are FALSE

op1 xor op2

logical xor (exclusive or) of values op1 and op2 - result is TRUE in case, that either op1 or op2 are TRUE

## Relational operators

Relational operators are used to compare values of the same type. It is possible to apply them on any simple or structured type. Method for evaluation operator applied to value list(s) is possible to specify using general mathematical qualifiers “exists“ and “for all“. Results of relational operator evaluations are logical values or their lists.

ToPas includes six common operators:

op1 = op2

op1 is equal op2

op1 <> op2

op1 is not equal op2

op1 < op2

op1 is less than op2

op1 <= op2

op1 is less or equal than op2

op1 > op2

op1 is greater than op2

op1 >= op2

op1 is greater or equal than op2

## String operators

String operator + is used to concatenate strings. Result of this string operator evaluation is also a string.

op1 + op2

concatenation of strings op1 and op2

## Date operators

Date operators are used to manipulate values of the date type. Result of addition (subtraction) of integer expressing number of days to (from) date is a date value. Result of subtraction date from date is an integer value expressing number of days. Leap years are properly calculated during expression evaluation, but the calendar reform in 1582 is not taken into account. So all date operators behave correctly in view of Gregorian calendar. Operands must represent correct date, otherwise the calculation of such an expression is interrupted.

op1 + op2

addition op2 days to date value op1

op1 - op2

subtraction of op2 days from date value op1

op1 - op2

day difference between date values op1 and op2 (subtraction date value op2 from op1)

## Typecasting operators

Typecasting operators make possible to change a value of one type into another one. Each operand in an expression can be attached with this operator, which is separated by colon symbol (:). This separator is followed by single letter specifying new type of value. Typecasting to string values enables us to

specify format of new string through minimum number of characters and number of decimal points. We can take advantage of this feature for output formatting in information forms.

**Typecasting to numeric values**

This operator changes an operand into an integer or real number value. Strings are always converted to real numbers. In case of need, Trunc or Round functions can be applied to obtain an integer value. If a string does not form a number, the calculation of such an expression is interrupted. Logical values are converted in following way: TRUE is converted into the integer value of 1 and FALSE into 0. Date values are converted into integers expressing the number of days starting from the 1st January of the year 1 in view of Gregorian calendar.

Typecasting to numeric values has following form:

op : N

**Example:** *'1234.12':N, @1.YEAR : N*

**Typecasting to string values**

Numeric values are converted into corresponding strings. Logical value TRUE is converted into string 'T' and FALSE into 'F'. Date values are converted into the form corresponding with date value constants. Typecasting to strings can be amended with optional format specifier, i.e. minimum number of characters in new string and number of decimal points (for real number values). If new string after value conversion is shorter than specified number of characters, result is affixed with leading spaces. In case of real numbers, number of characters is always at least 8 a maximum number of decimal points is 11. If number of decimal points is omitted, real numbers are converted into the exponential form, otherwise into a fixed-point decimal string.

Typecasting to string values has following form:

op : C : total_characters : decimal_points

**Example:** *1234.56:C:10:3,*
*@1.BUDGET:C:6:0,*
*@SALESDATE:C*

**Typecasting to logical values**

This operator converts all nonzero numeric values into the value TRUE and zero value into FALSE. Strings 'T' and 'TRUE' (regardless of letter case, leading and trailing spaces) are converted into the value TRUE, other strings into FALSE. Date values cannot be converted into logical ones.

Typecasting to logical values has following form:

op : L

**Example:** *@1.USED : L*

**Typecasting to date values**

Numeric values are converted into date values considering them as the number of days starting from the 1st January of the year 1 in view of Gregorian calendar. Strings are converted into date values, if they syntactically forms date constant. Logical values cannot be converted into date values.

Typecasting to date values has following form:

op : D

**Example:** *@1.DAY : D*

## Operator precedence

Usual rules of precedence clarify the order in which complex expressions are evaluated. Operators with higher priority are calculated earlier. Operators with equal precedence are always performed from left to right. Operands are evaluated prior to all operators.

Operator precedence from higher to lower:

1. typecasting operators
2. unary operators (+, -, not),
3. multiplying operators (*, /, div, mod, shl, shr, and),
4. adding operators (+, -, or, xor),
5. relational operators (=, <>, <, <=, >, >=).

**Example:**
*10-3*2*
** has higher precedence than - (result 4)*
*(10-3)*2*
*expression in parenthesis is treated as a single operand for multiplication (result is 14)*
*(@REGION = 'PRAGUE') and (@1.BUDGET > 1000)*
*expressions in parenthesis are operands for logical and*
*@REGION = 'PRAGUE' and @1.OBRATY > 1000*
***!** syntactically invalid expression - operator and has higher precedence than relational operators, so that its operand cannot be string value*

## Value list expressions

References to secondary (external) database tables, in fact, represent several values (list) of the same type. So that, lists are used during the evaluation of these references and the calculation is automatically performed for all possible results of such references. That is why structured types and other constructs (e.g. list selection or sort functions) for processing expressions representing value lists are included in ToPas.

## Evaluation of list expressions

ToPas imposes following limitation on such expression: only one operand of arithmetic, string or date operators can be an expression representing list of values. Similar rule applies to internal mathematical, string or date functions with more arguments. Such an operation is then performed for all values in the list and returns a list of corresponding results (number of results cannot increase).

**Example**: *@1.BUDGET/1000*
*all references to budget will divided by 1000*

Binary logical an relational operators enables us to combine two operands representing value lists. For more precise specification of their processing and result calculation, it is possible to use for each value list operand general mathematical qualifier “exists“ or “for all“.

## Using value list expressions

Expressions representing value lists can be used in all tools, where the expression can be specified.

If the result of an expression is a list of values, the first found result is used during display of labels for graphical objects or cartodiagrams as well as replace of database table item values. The order of evaluation is in that case system dependent. Queries for selection of graphical objects (or records in database table browser) are satisfied and an object (or record) is selected, if at least one result has the value TRUE. Query is not satisfied only and only if all result has the value FALSE or query returns no results.

Sometimes, we need to obtain only one value for an expression representing value list, e.g. for its next use in arithmetic operation. For this purpose, ToPas provides several constructs. Statistical functions provides the simplest way, which are sufficient for the most of cases.

**Example:** *Max(@1.BUDGET)*

It is also possible to forbid a database table join evaluation, if it represents more than one value (result). In that case, it is necessary to use exclamation mark symbol (!) instead of dot symbol (.) for the separation of references to database joins. If it is found during this reference evaluation more than 1 record attached, the calculation is interrupted and the result for this expression is not evaluated. It is

possible to take advantage of this feature especially for safe specification of the database table item replace.

**Example**: *@2!NAME*
*Shop name is found only in the case, when one name (database record) is attached to the shop object through join no. 2.*

Using indexes within full identification of the referenced table is another method to reduce amount of results for an expression representing value list. Index specification is in greater detail described in the section Extended references to database tables. Indexes correspond to the position in the record set attached to particular record, which ranges in 1..Num(reference). Index can be specified using an expression which returns an integer value. If it returns for example the value of 3, then the third of attached record is chosen.

The order of records attached to a particular item through a database join is system dependent. For the reason of correct evaluation of references to external database tables, it is necessary to specify indexes for all joins in full identification or the last join in identification, if we use indexes. It is clear from these limitations, that this construct is not intended for standard solutions and it is provided rather to trap exceptional situations.

**Example:**
*Table of month budgets (items MONTH, BUDGET) for each shop is linked through join no. 1. To find the month with maximum month budget, following expression can be used*
*@1.[NMax(@1.OBRATY)]MESIC*
*Position (index) of the maximum budget within records attached to a particular shop is found using the expression NMax(@1.OBRATY). This index is then used to retrive necessary month indication.*

## Mathematical qualifiers

Binary logical an relational operators enables us to combine two operands representing value lists. For more precise specification of their processing and result calculation, it is possible to use for each value list operand general mathematical qualifier **"exists"** or **"for all"**.

"Exists" qualifier is implicitly used in the case, when "for all" qualifier is not explicitly specified. Qualifier "for all " is denoted by ampersand symbol (&) on the corresponding operator side. & symbol has to precede or follow an operator without any space. Qualifier "for all" can not be used for operands which are not value lists. It is possible to use logical or relational operators in following form:

oper

&oper

oper&

&oper&

**Example for operator <=**: *<=, &<=, <=&, &<=&*

In the case of "exists" operator, all possible operand variants are calculated. It means, if the operand on the left side results in two values and the operand on the right side results in three values, we receive six values (if operands are independent each other in view of databases and their models).

**Example:**
*Table of month budgets (item BUDGET) for each shop is linked through join no. 1. To find all shops with one budget (at least) higher than 10,000, following expression can be used*
*@1.BUDGET > 10000*

If we specify "for all" qualifier for an operand, all results calculated for this operand variants are combined use logical and operator. It means, if the operand on the left side results in two values, the operand on the right side results in three values and "for all" qualifier is specified for the second operand, we receive two values (if operands are independent each other in view of databases and their models).

**Example:**
*Table of month budgets (item BUDGET) for each shop is linked through join no. 1. To find all shops with all budgets higher than 10,000, following expression can be used*

```
@1.OBRATY &> 10000
```

If an expression includes several references to the same database table or database join, the calculation respects tabular character of these data and the expression is correctly evaluated from that point of view. Operands containing references to the same table or join are calculated together, so that references to different records cannot be combined. It also means, that more references to the same table (or join) in one expression are always calculated for the same record (or join key).

**Example:**
*Table of month budgets (items MONTH, BUDGET) for each shop is linked through join no. 1. To find all shops with January budget higher than 1,000, following expression can be used*
*(@1.MONTH = 'JANUARY') and (@1.BUDGET > 1000)*
*For each shop, there is 12 records for each month in table. Left and right side of this expression returns 12 results. In normal situation, "exists" qualifier combines them into 144 results. In this case, we receive 12 results only - 11 results will have the value FALSE (values for other months) and one will have the value TRUE or FALSE according to the result of the right operand.*

**Note:** For the reason of correct evaluation of references to external database tables, it is not possible to combine in one expression (program) more than one different record in each primary database table within such references at a time. This rule is checked during expression evaluation. There exist workaround for ToPas programs, where we can use the same variable for all such references. In that case, it is possible to change this variable and access sequentially different records, though, in different expressions.

**Example:**

```
if @[%[L,I]RArea]1.ID = @[%[L,I]LArea]1.ID
            ^expression cannot be evaluated

NumAre := %[L,I]RArea; ID1 := @[NumAre]1.ID;
NumAre := %[L,I]LArea; ID2 := @[NumAre]1.ID;
if ID1 = ID2
            ^following sequence evaluates correctly
```

# Declaration parts

## Constant declaration

starts with the keyword const and contains declarations of individual constants. The declaration of each constant consists of constant identifier and value, which are separated by the equal symbol (=). Declarations of individual constants are separated by the semicolon symbol (;).

The constant declaration has following form:

```
const  ident1 = value1;
....
identN = valueN;
```

**Example:**

```
const  MaxCount = 30;
       Name = 'Town: ';
       DateLimit = 30.5.1995;
```

## Variable declaration

starts with the keyword var and contains declarations of individual variables of the same type. The declaration of variables of the same type consists of list of variable identifiers separated by the comma symbol (,) and type name. The type name is separated from the identifier list by the colon symbol (:). Individual variable declarations are separated by the semicolon symbol (;).

The variable declaration has following form:

```
var  ident11,...,ident1N : type1;
...
identN1,...,identNN : typeN;
```

**Example:**

```
var  Count, I: longint;
     CreationDate : datum;
     Budgets: list of longint;
     ShopNames: list of string;
     ShopCount: longint;
```

## Declaration of alias names for database tables

starts with the keyword database and contains declarations of individual aliases. The declaration of each alias consists of alias name identifier and database table filename in string form, which are separated by the equal symbol (=). The database table filename is specified either by its full path or relatively to TopoL working directory, which is defined during system start-up. Declarations of individual aliases are separated by the semicolon symbol (;).

In TopoL, the predefined identifier CURRENT can be used for current database table, i.e. database table, which is processed, or primary database table in current block attached to object type and group, for which ToPas expression (program) is calculated.

The database table alias declaration has following form:

```
database  ident1 = jmeno_soubor1;
...
identN = jmeno_souborN;
```

**Example:**

```
database
  Budgets = 'BUDGETS.DBF';
  Turnover = 'C:|ACCOUNT|TURNOVER.DBF';
```

## Declaration of alias names for blocks of graphical objects

starts with the keyword block and contains declarations of individual aliases. The declaration of each alias consists of alias name identifier and block directory in string form, which are separated by the equal symbol (=). Block directory is specified either by its full path or relatively to TopoL working directory, which is defined during system start-up. Declarations of individual aliases are separated by the semicolon symbol (;).

In TopoL, the predefined identifier CURRENT can be used for current block, i.e. block containing objects, for which ToPas expression (program) is calculated.

The block alias declaration has following form:

```
block ident1 = adresar1;
...
identN = adresarN;
```

**Example:**

```
block
  TOWNS = 'TOWNS.BLK';
  OurMap = 'C:|MAP|CADASTER.BLK';
```

# Statements

## Assignment statement

Assignment statement consists of variable reference, which value is assigned to, and expression, which evaluation results in assigned value. Both statement parts are separated by the assignment operator (symbol :=). The expression must be assignment-compatible with the type of variable reference.

The assignment statement has following form:

var_ref := expression

**Example:**

```
Count := Num(@1.BUDGET)
Budgets := @1.BUDGET
Budgets[1] := Budgets[1]*2.54
```

## Output procedures

Output procedures are similar to the text output in Pascal. Procedures are identified by the keywords write and writeln, which are followed by list of expressions enclosed by parenthesis. The writeln procedure does the same as write and then writes an end-of-line marker to the output. In case of information forms, the output device is form window. These procedures create the only results of information form processing, which are displayed in this window.

The output procedures have following forms:

write ( expression1, ..., expressionN )

writeln

writeln ( expression1, ..., expressionN )

**Example:**

```
write('Count: ',Count,
      'Budget sum: ',Sum(@1.BUDGET))
writeln
```

## Compound statement

Compound statement enables us to insert several statements into constructs, where single statement is expected. These statements are enclosed by the keywords begin and end. Individual statements are separated by semicolons (;).

The compound statement has following form:

begin statement1; ... statementN end

**Example:**

```
begin
  Count := Num(@1.BUDGET);
  write('Count: ',Count,
        'Budget sum: ',Sum(@1.BUDGET))
  writeln
end
```

## Conditional statement

Conditional statement is similar to Pascal. It starts with the keyword if, which is followed by boolean type expression. If the expression produces the value TRUE, then the statement following keyword then is executed. If the expression produces FALSE, then the statement following keyword else is executed (if this optional part is present). Because the form of conditional statement can raise syntactic ambiguity, applies the rule that else part is associated with the preceding then part. Therefore the use of compound statements is strongly recommended in nested conditional statements.

The conditional statement has following forms:

if expression then statement

if expression then stat1 else stat2

**Example:**

```
if Num(@1.BUDGET) > 0
then begin
  writeln(' Budget sum: ',
                Sum(@1.BUDGET))
end
else begin
  writeln(' No budget information'
                + ' in this unit');
end
```

## Repetitive statements

Repetitive statements has the same syntax and semantics like in Pascal. While loop is executed as long as condition is true. Repeat loop is executed until condition becomes true. Condition is a boolean type expression. Number of for loop repetitions is defined by limit values (ini_ex and fin_ex). The control variable id_var and both limit expressions must be of the integer type. The value of the control variable is incremented (for keyword to) or decremented (for keyword downto) by one for each repetition.

The repetitive statements have following forms:

while condition do statement

repeat stat1; ... statN until condition

for id_var:=ini_ex to fin_ex do statement

for id_var:=ini_ex downto fin_ex do statement

**Example:**

```
while (I <= Count)
do begin
  writeln;
  I := I+1;
end

repeat
  writeln;
  I := I+1;
until (I > Count)

for N := I to Count
do writeln
```

# Colour and symbol format

## Colour number format for vector data

Colour of graphical objects stored in data is expressed in the internal TopoL format. Colour in the internal format is integer value represented by two bytes (16 bits). Values are divided into two colour groups:

Positive colour numbers in range 0..15 denote standard colours of vector data, which are listed in the following table.

| Number and colour | Number and colour |
|---|---|
| 0 - light grey | 8 - light blue |
| 1 - blue | 9 - light green |
| 2 - green | 10 - light cyan |
| 3 - cyan | 11 - light red |
| 4 - red | 12 - light magenta |
| 5 - magenta | 13 - black |
| 6 - brown | 14 - yellow |
| 7 - dark grey | 15 - white |

Negative colour numbers make possible to specify any colour from the set of 32767 colours represented in modified RGB model (for 15 bits).

Other numbers (positive ones greater than 15) are currently invalid for the colour specification in the internal format.. Functions for colour manipulations are able to translate the internal format into standard colour format in the MS Windows environment and control the standard colour format (colour field intensity value extraction, colour specification based on colour field intensity values).

## Symbol number format for vector data

Symbol of graphical objects stored in data is expressed in the internal TopoL format. Symbol in the internal format is integer value represented by two bytes (16 bits). Values are divided into two symbol groups:

Symbol numbers in range 0..999 denote standard symbols (point symbols, line or fill styles) of vector data. The predefined standard symbols currently don’t fully use this range.

Symbol numbers greater than 1000 make possible to specify user-defined symbols.

Other numbers (negative ones) are currently invalid for the symbol specification in the internal format.