

MiniCalc

Content

1 MiniCalc 1

1.1 Overview 1

1.2 Example 1

1.3 Comments 2

1.4 Operators and Operands 2

1.5 Environments and Variables 3

1.6 Functions 4

1.6.1 Built-In Functions 4

1.6.2 Aggregate Functions 11

1.6.3 Defining New Functions in Delphi 12

1.6.4 Defining New Functions in MiniCalc 13

1.7 Nested Blocks 15

1.8 Control Structures 16

1.9 Objects 16

1.10 Built-In Constants 17

1.11 Filters, Comparators, and Transforms 18

2 Index 21

1 MiniCalc

1.1 Overview

MiniCalc is an expression parser and evaluator written in Delphi. You can use MiniCalc to make an application more flexible and dynamic.

MiniCalc is implemented in the DSMiniCalc unit. There are three steps to using MiniCalc:

1. Parse an expression.
2. Create an environment, which holds variable names and their associated values.
3. Evaluate the parsed expression in the environment.

Once you have parsed an expression, you can evaluate it any number of times, without re-parsing it.

MiniCalc includes common operators, such as addition, subtraction, multiplication, division, exponential, logical, bit-wise, etc. It also has numerous built-in functions and allows you to define your own functions (implemented in Delphi).

MiniCalc has a few built-in constants, and allows you to define your own.

Of course, MiniCalc supports variables, including objects that have named properties.

Additional topics:

- Brief introduction to MiniCalc.
- Simple example of using MiniCalc.

1.2 Example

Example

The following example illustrates how to use MiniCalc to evaluate a simple expression. This code evaluates $4 + 5$ and prints the result, which, of course, is 9.

```
uses
  DSMiniCalc;

procedure RunMain;
var
  Expr: ICalcExpr;
  V: Variant;
begin
  Expr := ParseCalcExpr('4+5');
  V := Expr.Evaluate(nil);
  Writeln(V);
end;
```

Discussion

The above example shows how to parse an expression, using *ParseCalcExpr*, and how to evaluate an expression, using the *Evaluate* method.

This example does not show how create and use an environment, which is necessary when the expression contains variables. We will discuss environments later. For now, when we evaluate an expression, we will pass nil for the environment parameter.

The *ICalcExpr* interface represents an expression that has been parsed by MiniCalc. The full definition of this interface is as follows:

```
ICalcExpr = interface ['{AB052841-0C95-11D5-8640-005004E91334}']
  function Evaluate(const Env: ICalcEnv): Variant;
  function AsString(const Env: ICalcEnv): string;
  function AsBool(const Env: ICalcEnv): Boolean;
end;
```

These three methods essentially do the same thing (they evaluate the expression). The only difference is in how the result is returned to you. Evaluate is the most flexible: it returns the result as a Variant. *AsString* and *AsBool* return the result as a string or a Boolean, respectively. We will discuss the *Env* parameter shortly.

1.3 Comments

Comments

MiniCalc supports two kinds of comments: line comments and block comments. Line comments begin with two backslashes (just as in Delphi). Block comments begin with "(" and are terminated with "**").

1.4 Operators and Operands

Operators and Operands

MiniCalc supports various operators that work on integers, floating-point values, strings, and Boolean values.

Operands

Internally, MiniCalc uses variants for everything. Integers and floating-point values are expressed as they would be in Delphi. Any number that cannot fit into an Integer will be converted to a Double. Scientific notation is supported. Strings are expressed as in SQL: you can use single-quotes or double-quotes to define a string. Within a string, you can escape the quote character by repeating it. Boolean values are expressed using the keywords `<tt>>true</tt>` and `<tt>>false</tt>`.

Null values are expressed using the global variable *@Null*.

Functions are described below.

Operators

MiniCalc has a rich set of operators, borrowing ideas from Delphi, C, and FORTRAN. The table below describes the operators defined in MiniCalc.

Operators	Notes	Precedence
()	Sub-expression grouping; function call	1
true false	Boolean constants	1
not	Unary logical negation	2
bitnot	Unary bit-wise negation	2
+ -	Unary plus or minus	2
.	Field selector (for objects)	3
^	Power (raise one number to the power of another)	4
* / div mod	Multiplication and division	5
shl shr	Bit-wise shift	5
+ -	Addition and subtraction	6
< > <= >=	Relational comparisons	7
= <>	Equality comparisons	8
and	Logical conjunction	9
bitand	Bit-wise conjunction	9

or xor	Logical disjunction	10
bitor bitxor	Bit-wise disjunction	10
? :	Conditional	11
:=	Assignment	12
,	Multiple expression list	13
var begin end	Nested statement block (with local variables)	14
;	Multiple statement list	15

Notes

Keywords (such as "div" and "true") are *not* case sensitive.

A lower precedence number means higher precedence. In general, MiniCalc attempts to emulate Delphi's operators. Notes on a few exceptions follow.

- The logical operators (**not**, **and**, **or**, and **xor**) require Boolean values. To perform bit-wise operations, use the bit-wise operators (**bitnot**, **bitand**, **bitor**, and **bitxor**). MiniCalc uses Boolean short-circuit evaluation for **and** and **or**.
- The ^ operator raises one number to the power of another. The result is always a Double. Internally, this operator uses Delphi's *Power* or *IntPower* functions.
- The ? : operator works as in C/C++/Java/C#.
- The . (dot) operator evaluates a named property of an object (see below for more details).
- The := (colon-equals) operator requires a modifiable expression on the left-hand-side. This includes variables and expressions involving the . (dot) operator.
- The , (comma) operator makes a compound expression out of two sub-expressions, evaluating them in sequence (just like in C/C++/C#).
- The **var**, **begin**, and **end** keywords are used to define local variables in a nested block of statements. See Nested Blocks for details.
- The ; (semi-colon) operator separates multiple expressions in sequence (much like the comma operator in C/C++/C#). The semi-colon may not appear inside a sub-expression, but a comma may.

1.5 Environments and Variables

Environments

MiniCalc employs the concept of an environment, to support variables within expressions. An environment is basically a map from name (a string) to a variant value. Setting a variable to the Unassigned variant value will remove the variable from the map. Similarly, evaluating an undefined variable will yield Unassigned. The evaluation methods of *ICalcExpr* require an environment object, which is of type *ICalcEnv*. This interface is defined as follows:

```
ICalcEnv = interface ['{AB052840-0C95-11D5-8640-005004E91334}']
  function GetValue(const Name: string): Variant;
  procedure SetValue(const Name: string; const Value: Variant);
  procedure Clear;
  function Map: IMap;

  property Values[const Name: string]: Variant
    read GetValue write SetValue; default;
end;
```

To create a new environment, use the *NewCalcEnv* function:

```
function NewCalcEnv(
  const Map: IMap;
  CaseSensitivity: TCaseSensitivity): ICalcEnv; overload;
```

```
function NewCalcEnv(
  CaseSensitivity: TCaseSensitivity): ICalcEnv; overload;
```

The first version creates a new environment using the supplied map and case sensitivity mode. The second version creates a new environment using a newly created map.

Here is an example of using an environment to take advantage of variables:

```
procedure RunMain;
var
  Env: ICalcEnv;
  Expr: ICalcExpr;
  V: Variant;
begin
  Env := NewCalcEnv(csIgnoreCase);
  Env['n'] := 5;
  Expr := ParseCalcExpr('2^n');
  V := Expr.Evaluate(Env);
  Writeln(V);
end;
```

The above example creates an environment, defines a variable named *n* (giving it the value 5), then computes 2^n , which is 32.

The Global environment

MiniCalc defines a global environment, which is an environment that overrides the environment passed into *Evaluate* (hereafter called the local environment). To resolve a variable reference, MiniCalc looks first in the global environment. If the variable is not defined there, MiniCalc will then look in the local environment. The global environment allows you to define new global variables and functions to be used throughout your program. You can manipulate the global environment from Delphi code via the following function:

```
function MiniCalcGlobals: ICalcEnv;
```

For example, you can define a new global variable as follows:

```
MiniCalcGlobals['CompanyName'] := 'First Trust Portfolios, L.P.';
```

The assignment operator may be used to define new variables into the local environment. If there is no local environment, then the global environment will be used. Also, within a MiniCalc expression, the *@SetGlobal* function can be used to define global variables (even if there is a local environment).

1.6 Functions

MiniCalc functions work similar to Delphi functions, with one notable exception: parenthesis are required. For example, to compute the square root of 2...

```
ParseCalcExpr('@Sqrt(2)');
```

If a function requires multiple arguments, separate them with commas. For example:

```
ParseCalcExpr('@Max(a, b)');
```

1.6.1 Built-In Functions

All built-in functions begin with an @-sign, to avoid colliding with object fields (discussed later). The following table describes the built-in functions. Many functions correspond directly to a standard Delphi function of the same name; these functions are simply marked with an asterisk (*) under the Notes column.

Function	Args	Notes
@Abs	1	*
@Acos	1	*

@Acosh	1	*
@Acot	1	*
@Acoth	1	*
@Acsc	1	*
@Acsch	1	*
@Add	2+	The first argument is a list (<i>IList</i> , <i>IIntfList</i> , or <i>IStringList</i>), the remaining arguments are added to the list, in order.
@AddPaths	2+	Concatenates two or more strings, using the <i>AddPaths</i> function defined in <i>DStringUtil</i> .
@Aggregate	2-3	Computes an aggregate expression over a container (array, vector, or iterator). See below for details.
@ArrayOf	1+	Constructs a variant array of variant values.
@ArrayOfType	2+	Constructs a variant array (the first argument specifies the element type).
@AsVect	1	Converts a value to an <i>IVector</i> . If the argument is already an <i>IVector</i> , then this is returned. Otherwise, a new <i>IList</i> will be returned, with contents depending on the argument. If the argument is a string, the list will contain one element per character in the string. If the argument is an <i>IAssociation</i> , the list will contain (in no particular order) key/value pairs (in the form of <i>IPropertyBag</i> objects with <i>Key</i> and <i>Value</i> properties).
@Asec	1	*
@Asech	1	*
@Asin	1	*
@Asinh	1	*
@Atan	1	*
@Atanh	1	*
@Avg	1	An aggregate function, computes the average of all non-Null values.
@Break	0	*
@Cast	2	Returns the first argument, typecast into the type indicated by the second argument.
@Ceil	1	*
@Ceil64	1	Like <i>@Ceil</i> , but returns an <i>Int64</i> (not available in Delphi5).
@Chr	1	*
@Continue	0	*
@CompareC	2	Case sensitive comparison (returns a number whose sign reflects the outcome of the comparison).

@Comparel	2	Case insensitive comparison (returns a number whose sign reflects the outcome of the comparison).
@Copy	3	Works like the Delphi function of the same name; supports strings, variant arrays, and the following interfaces: <i>IIntfList</i> , <i>IStringList</i> , <i>IList</i> , and <i>IVector</i> (the copy will be an <i>IList</i> object).
@Cos	1	*
@Cosh	1	*
@Cot	1	*
@Coth	1	*
@Count	1	An aggregate function, computes the number of non-Null values.
@Csc	1	*
@Csch	1	*
@Date	0-1	The zero-argument version works like Delphi's <i>Date</i> function; the one-argument version works like Delphi's <i>DateOf</i> function.
@Defined	1+	Returns False if any argument is unassigned; returns True otherwise.
@DSSimpleRoundTo	2	Returns <i>DSMathUtil.DSSimpleRoundTo</i> .
@Error	1	Raises an exception, with the specified message.
@Equal	2	Returns True if the two arguments are equal (using the <i>VariantEqual</i> function defined in <i>DSGenUtil</i>).
@Evaluate	1	The argument must either be a string or the result of calling <i>@Expr</i> or <i>@Parse</i> . For a string, <i>@Evaluate</i> will parse the string as a MiniCalc expression, then evaluate it. On the other hand, if the argument is an <i>ICalcExpr</i> (as returned by <i>@Expr</i> or <i>@Parse</i>), then it is evaluated in the current environment.
@Exp	1	*
@Exit	0	*
@Expr	1	Takes one argument, but doesn't evaluate it. Instead, returns an object that may be passed to <i>@Evaluate</i> for evaluation.
@ExtractFileName	1	*
@ExtractFilePath	1	*
@ExtractFileExt	1	*
@ExtractFileBase	1	Like the function with the same name defined in <i>DSStringUtil</i> .
@FileExists	1	*
@FindFirst	2-3	Searches a collection for the first element matching a filter condition. The first argument is the collection (string, array, or <i>IVector</i>). The last argument is the filter expression (evaluated for each item until a match is found). Within this expression, the item to be tested is available as <i>@Item</i> . However, if 3 arguments are given, the second argument specifies the variable to use instead of <i>@Item</i> . For example: @FindFirst('abcxyz', ch, ch >= 'm') (returns 'x'). If no item matches the filter, <i>Unassigned</i> is returned.

@Floor	1	*
@Floor64	1	Like @Floor, but returns an Int64 (not available in Delphi5).
@For	3-4	Deprecated. Implements a for loop, as in C++. @For(Init, Test, Update, Body) -- equivalent to the following C++ code: for (Init; Test; Update) Body MiniCalc now supports the C++/Java style for loop directly, so the @For function should be avoided.
@Format	1+	*
@FormatDateTime	2	*
@Frac	1	*
@GetElem	2	The first argument is a collection (string, array, or <i>IVector</i>), the second argument is an integer index. This function returns the specified element of the collection. Note that strings are indexed starting at 0, not 1.
@GetKeys	1	Takes an <i>IAssociation</i> and returns an <i>IList</i> containing all keys in the association (in no particular order).
@GetNamePart	1-2	Like the function in <i>DSSStringUtil</i> of the same name.
@GetValue	2	The first argument is an <i>IAssociation</i> interface (defined in <i>DSGenUtil</i>), the second argument is a property name (string). This function returns the property of the association.
@GetValues	1	Takes an <i>IAssociation</i> and returns an <i>IList</i> containing all values in the association (in no particular order).
@GetSubValue	2	Works like @GetValue, but supports sub-properties with the dot separator.
@GetValuePart	1-2	Like the function in <i>DSSStringUtil</i> of the same name.
@GlobalEnv	0	Returns the global MiniCalc environment, as an <i>IAssociation</i> .
@HasValue	2	The first argument is an <i>IAssociation</i> interface (defined in <i>DSGenUtil</i>), the second argument is a property name (string). This function returns True if the association has the specified property.
@HasSubValue	2	Works like @HasValue, but supports sub-properties with the dot separator.
@High	1	Returns the high index of a string, array, or <i>IVector</i> (good for loops).
@Hypot	2	*
@IfBlank	2	Returns the first argument, unless is blank (as defined by @IsBlank), in which case the second argument is returned.
@IfNull	2	Returns the first argument, unless it is Null, in which case the second argument is returned.
@IndexOf	2-3	Like @FindFirst, but returns the index of the matching item (or -1 if not found).
@InC	2+	Returns True if the first argument is equal to any of the other arguments (case sensitive). Similar to the SQL "in" operator.
@InI	2+	Returns True if the first argument is equal to any of the other arguments (ignoring case). Similar to the SQL "in" operator.

@Inspect	1	Converts the argument into a string, in a format that is appropriate for debugging (using the <i>MiniCalcInspect</i> function).
@Int	1	*
@Int64	1	Converts the argument into an Int64 value (valid in Delphi 6 and up).
@IsArray	1	Returns True if the argument is a variant array.
@IsAssoc	1	Returns True if the argument is an <i>IAssociation</i> interface (defined in <i>DSGenUtil</i>). If so, the argument is compatible with the <i>@GetValue</i> , <i>@HasValue</i> , and <i>@SetValue</i> functions.
@IsBlank	1	Returns True if the argument is Null, Unassigned, or an empty string.
@IsExpr	1	Returns True if the argument is a parsed expression obtained by <i>@Expr</i> or <i>@Parse</i> .
@IsFunc	1	Returns True if the argument is a function (user-defined or built-in).
@IsInfinite	1	*
@IsIterator	1	Returns True if the argument is an <i>IIterator</i> object.
@IsList	1	Returns True if the argument is an <i>IList</i> , <i>IIntfList</i> , or <i>IStringList</i> object.
@IsMap	1	Returns True if the argument is an <i>IMap</i> or <i>IIntfMap</i> object.
@IsNaN	1	*
@IsNull	1	Returns True if the argument is Null.
@IsObject	1	Returns True if the argument is a MiniCalc object.
@IsPersist	1	Returns True if the argument is an <i>IPersistObject</i> interface (defined in <i>DSSStreams</i>).
@IsPrime	1	Returns <i>DSMathUtil.IsPrime</i> .
@IsVect	1	Returns True if the argument is an <i>IVector</i> interface (defined in <i>DSGenUtil</i>). If so, the argument is compatible with the <i>@GetItem</i> and <i>@HasItem</i> functions.
@IsZero	1	*
@Length	1	Returns the length of a string, array, or <i>IVector</i> object.
@Ln	1	*
@LocalEnv	0	Returns the current MiniCalc environment, as an <i>IAssociation</i> .
@Log10	1	*
@Log2	1	*
@LogN	1	*
@Low	1	Returns the low index of a string, array, or <i>IVector</i> (good for loops).
@LowerCase	1	*
@LoadIniFile	1	Opens the specified INI file, and returns an <i>IPropertyBag</i> containing the contents of the INI file.

@Min	1-2	Returns the smaller of the two arguments. The single-argument version works the same as <i>@MinAgg</i> .
@MinAgg	1	An aggregate function, computes the smallest non-Null value.
@Max	1-2	Returns the larger of the two arguments. The single-argument version works the same as <i>@MaxAgg</i> .
@MaxAgg	1	An aggregate function, computes the largest non-Null value.
@NewBufInStream	1-2	Creates a MiniCalc object wrapper around a new <i>IBufInStream</i> . The first argument is the input stream; the second argument, if present, is the buffer size.
@NewBufFileInStream	1-2	Creates a MiniCalc object wrapper around a new <i>IBufInStream</i> . The first argument is the name of the file to open; the second argument, if present, is the buffer size. Methods include: <i>ReadString</i> , <i>ReadIn</i> , and <i>Eof</i> .
@NewFileInStream	1	Creates a MiniCalc object wrapper around a new <i>IInputStream</i> . The first argument is the name of the file to open. Methods include: <i>ReadString</i> .
@NewFileOutputStream	1	Creates a MiniCalc object wrapper around a new <i>IOutputStream</i> . The first argument is the name of the file to open.
@NewIntfList	0	Returns a new <i>IIntfList</i> object.
@NewIntfMap	4	Returns a new <i>IList</i> object; arguments: <i>Duplicates</i> , <i>Threading</i> , <i>TableSize</i> , <i>CaseSensitivity</i> .
@NewList	0-2	Returns a new <i>IList</i> object; arguments: <i>CaseSensitivity</i> , <i>ReallocPercent</i> .
@NewMap	4	Returns a new <i>IMap</i> object; arguments: <i>Duplicates</i> , <i>Threading</i> , <i>TableSize</i> , <i>CaseSensitivity</i> .
@NewPersist	1	Returns a new <i>IPersistObject</i> object; argument: <i>ObjectName</i> .
@NewPropBag	0	Returns a new <i>IPropertyBag</i> object.
@NewStringList	0	Returns a new <i>IStringList</i> object.
@NextPrime	1	Returns <i>DSMathUtil.NextPrime</i> .
@NiceDateTime	1	Like the function of the same name in <i>DSStringUtil</i> .
@NiceDate	1	Like the function of the same name in <i>DSStringUtil</i> .
@NotDefined	1	Returns True if the argument is unassigned.
@NotEqual	2	Returns True if the two arguments are not equal (using the <i>VariantEqual</i> function defined in <i>DSGenUtil</i>).
@NotNull	1+	Returns False if any argument is Null, or True if none are Null.
@Now	0	*
@Parse	1	Parses the argument, a string, into a MiniCalc expression, which can be evaluated subsequently via <i>@Evaluate</i> .
@ParseCSV	4	Like the function of the same name in <i>DSStringUtil</i> . The last 3 arguments are strings. The function returns an <i>IStringList</i> .
@Pos	2	*

@PrintPropBag	1-2	Converts an <i>IPropertyBag</i> object to a string, using <i>DSPROPERTYBag.PrintPropBag</i> . The optional second argument specifies a name for the property bag.
@Raise	1	Raises an exception with the specified value (normally a string).
@Return	1	Exits the current function (similar to <i>@Exit</i>) with the specified return value.
@Reverse	1	Returns a reversed copy of the given string, array, <i>IList</i> , or <i>IStringList</i> object.
@Root	2	Returns the <i>n</i> th root of <i>n</i> ; arguments: <i>n</i> , <i>x</i> . For example, @Root(3, 8) means the cubed root of 8, which is 2.
@Round	1	*
@RoundToDigits	2	Returns <i>DSMathUtil.RoundToDigits</i> .
@SameObject	2	Returns <i>DSGenUtil.SameObject</i> .
@SameText	2	*
@Sec	1	*
@Sech	1	*
@SetElem	3	Takes three parameters: <i>V</i> , <i>i</i> , <i>x</i> . <i>V</i> is an <i>IVector</i> interface (defined in <i>DSGenUtil</i>), <i>i</i> is an integer index, and <i>x</i> is any value. This function sets the <i>i</i> th element of <i>V</i> to <i>x</i> . In other words, $V[i] := x$.
@SetGlobal	2	Sets a global variable. The first argument is the name (a string) of the global variable, and the second argument is the desired value.
@SetValue	3	Takes three parameters: <i>A</i> , <i>s</i> , <i>x</i> . <i>A</i> is an <i>IAssociation</i> interface (defined in <i>DSGenUtil</i>), <i>s</i> is a string, and <i>x</i> is any value. This function sets the property named <i>s</i> of <i>A</i> to <i>x</i> . In other words, $A[s] := x$.
@SetSubValue	3	Works like <i>@SetValue</i> , but supports sub-properties with the dot separator.
@Sign	1	*
@SimpleRoundTo	2	*
@Sin	1	*
@Sinh	1	*
@Sqr	1	*
@Sqrt	1	*
@StringReplace	3-5	*
@Sum	1	An aggregate function, computes the sum of all non-Null values.
@Supports	2	Arguments: <i>Instance</i> , <i>IID</i> . Attempts to cast the instance to the specified interface. Returns nil if the interface is not supported.
@Tan	1	*
@Tanh	1	*

@Time	0-1	The zero-argument version works like Delphi's <i>Time</i> function; the one-argument version works like Delphi's <i>TimeOf</i> function.
@Trim	1	*
@TrimLeft	1	*
@TrimRight	1	*
@Unassigned	0	Always returns the Unassigned Variant value. Note that @Nil and @Null are global variables, but @Unassigned has to be a function (because the global environment does not store Unassigned values).
@UpperCase	1	*
@VarType	1	*
@While	2	Deprecated. Implements a while loop. Arguments: <i>Condition</i> , <i>Body</i> . MiniCalc directly supports the while loop, so this function should be avoided.

(*) These functions correspond directly to the standard Delphi function of the same name.

It is important to note that a function call does not necessarily cause all of its arguments to be evaluated. For example, @IfNull does not evaluate the second argument unless the first one is Null. Similarly, @InI and @InC evaluate stop evaluating arguments as soon as a match is found.

1.6.2 Aggregate Functions

MiniCalc includes support for special aggregate functions, which work like their counterparts in SQL. The aggregate functions include:

- @Avg- the average of all non-Null values encountered.
- @Count- the number of non-Null values encountered.
- @Max- the largest non-Null value encountered.
- @Min- the smallest non-Null value encountered.
- @Sum- the sum of all non-Null values encountered.

Each of these functions takes a single argument (the two-argument forms of @Min and @Max are not aggregates). If they appear in an expression that is evaluated multiple times in the same environment, then they will compute the appropriate aggregate value.

During the aggregation process, two special variables are defined: @Collection and @Item, which correspond respectively to the collection being traversed and the current item within that collection.

The @Aggregate function provides support for the above aggregate functions in MiniCalc. @Aggregate takes two or three arguments:

1. **Container:** The container (*varaint array*, *IVector*, or *IIterator*) to traverse,
2. **Expression:** An expression containing aggregate functions,
3. **Filter:** An optional Boolean expression used to filter items out of the aggregation.

@Aggregate traverses every element of the container, which must be a list or an array, and evaluates the expression for each item that matches the filter. If the filter is not specified, then every item in the container is processed. For example:

```
var
  A: Variant;
  Env: ICalcEnv;
```

```

begin
  Env := NewCalcEnv(csIgnoreCase);

  A := VarArrayCreate([0, 3], varVariant);
  A[0] := 1;
  A[1] := 2;
  A[2] := 3;
  A[3] := 4;
  Env['A'] := A;

  Writeln(ParseCalcExpr('@Aggregate(A, @Sum(@Item))').Evaluate(Env));
  // Prints '10'

  Writeln(ParseCalcExpr('@Aggregate(A, @Count(@Item), @Item > 1)').Evaluate(Env));
  // Prints '3'
end;

```

In order to use aggregate functions directly (*i.e.*, without using the `@Aggregate` function), you must use a special kind of environment. The functions `AggregateVector`, `AggregateIterator`, and `AggregateArray` takes care of this for you. Use `AggregateVector` with `IVector` collections, `AggregateIterator` with iterators, and `AggregateArray` with variant arrays. For example:

```

var
  Customers: IList;
  Customer: IPropertyBag;
  Count: Integer;
begin
  Customers := NewList(csIgnoreCase);

  // Add items to the Customers collection...
  Customer := NewPropertyBag;
  Customer['Address'] := '123 Main St.';
  Customers.Add(Customer);

  Customer := NewPropertyBag;
  Customer['Address'] := Null;
  Customers.Add(Customer);

  // Count how many have a non-Null Address property...
  Count := AggregateVector(
    Customers, nil, ParseCalcExpr('@Count(Address)'), nil);

  Writeln(Count);
end;

```

As this example illustrates, you can simply pass an `IVector` object to the `AggregateVector` function. This function requires the following

1.6.3 Defining New Functions in Delphi

There are two ways to define new functions to use in MiniCalc expressions: they can be implemented in Delphi or in MiniCalc itself. This section discusses how to define MiniCalc functions in Delphi. The next section discusses defining MiniCalc functions in MiniCalc itself.

User-defined functions are simply variables that hold a function object (as opposed to normal variables that might hold an integer or a string). Therefore, to define a new function, you only need to create the function object. Then you can define a variable with that value.

There are two ways to define a function object. One way is to implement the `ICalcFunc` interface:

```

ICalcFunc = interface ['{E94748C1-B43F-11D5-8640-005004E91334}']
  function Evaluate(
    const Env: ICalcEnv; const Args: ICalcExprList): Variant;
end;

```

The `Env` parameter provides a reference to the environment in which the function is executing. The `Args` parameter is a list of `ICalcExpr` objects that make up the arguments to the function.

A simpler approach is to use one of the *NewCalcFunc* functions:

```
TMiniCalcFunc = function(  
  const Env: ICalcEnv; const Args: ICalcExprList): Variant;  
  
TMiniCalcMeth = function(  
  const Env: ICalcEnv; const Args: ICalcExprList): Variant of object;  
  
function NewCalcFunc(Func: TMiniCalcFunc): ICalcFunc; overload;  
  
function NewCalcFunc(Meth: TMiniCalcMeth): ICalcFunc; overload;
```

As you can see, one version of *NewCalcFunc* takes a function pointer; the other takes a method pointer. Here is a complete example. First, we implement a function to safely divide two numbers (avoiding the zero divide exception)...

```
function SafeDivide(  
  const Env: ICalcEnv; const Args: ICalcExprList): Variant;  
var  
  v1, v2: Variant;  
begin  
  Args.CheckCount('SafeDivide', 2);  
  v1 := Args[0].Evaluate(Env);  
  v2 := Args[1].Evaluate(Env);  
  if v2 = 0 then  
    Result := v2  
  else  
    Result := v1 / v2;  
end;
```

Note two things about using the *Args* collection:

- The *CheckCount* method validates that the number of arguments is correct.
- When evaluating arguments, we must be sure to pass along the environment.

Now we can use this function in MiniCalc...

```
procedure RunMain;  
var  
  Env: ICalcEnv;  
  Expr: ICalcExpr;  
  V: Variant;  
begin  
  Env := NewCalcEnv(csIgnoreCase);  
  Env['SafeDivide'] := NewCalcFunc(SafeDivide);  
  Env['num'] := 5;  
  Env['denom'] := 0;  
  Expr := ParseCalcExpr('SafeDivide(num, denom)');  
  V := Expr.Evaluate(Env);  
  Writeln(V);  
end;
```

If you define any functions (or other variables) into the global environment, please remember to use the @-sign prefix, to avoid colliding with user-defined variables and object fields (discussed next). Functions and variables defined in a local environment do not need to follow this convention (as in our example above).

1.6.4 Defining New Functions in MiniCalc

You may define new MiniCalc functions in MiniCalc itself, using the *function* keyword. The syntax is very similar to Delphi. The following example illustrates how to define a function that returns doubles a number and then adds one:

```
function MyFunc(x);  
begin  
  2*x + 1;  
end;
```

The following function returns the sum of all integers in the range Low..High:

```
function SumRange(Low, High);
var
  i, Total;
begin
  Total := 0;
  for i := Low to High do
    Total := Total + i;
  @Return(Total);
end;
```

Functions are values, so they may be used like other data types. For example, to make an alias for the `SumRange` function, simply set another variable to its value:

```
MySum := SumRange;
```

The built-in function `@Inspect` automatically converts user-defined functions (created with the `function` keyword) into strings, for debugging purposes.

```
@ShowMessage(@Inspect(SumRange));
```

It is possible to define anonymous functions, which are functions without a name. You will typically either call an anonymous function directly, or pass it to another function. To define an anonymous function, enclose the function declaration in parenthesis, and omit the function name. The following example defines an anonymous function and then calls it.

```
(function; begin @ShowMessage('hello'); end)();
```

The following example passes an anonymous function to another function.

```
function ForEach(v, f);
var
  i;
begin
  for i := 0 to @High(v) do
    f(@GetElem(v, i));
  end;

  ls := @NewStringList();
  @Add(ls, 'a', 'b', 'c', 'd');

  ForEach(ls, (function(x); begin @ShowMessage(x); end));
```

MiniCalc offers an advanced feature for function parameters called "pass-by-expression". When a parameter is passed by expression, the actual argument is not evaluated at the point of call. Instead the expression is passed to the function, which can then evaluate it via the standard `@Evaluate` function. Pass-by-expression arguments are prefixed with a question mark, as in the following example.

```
function IfThenElse(Test, ?Yes, ?No);
begin
  if Test then
    @Evaluate(Yes)
  else
    @Evaluate(No);
  end;

  IfThenElse(x > 4, f(x), g(x));
```

The `IfThenElse` function only evaluates one of the two pass-by-expression arguments. Thus, in the call to `IfThenElse` above, only one of the functions `f` and `g` will be invoked. This is equivalent to the following:

```
x > 4 ? f(x) : g(x)
```

which is also the same as `if x > 4 then f(x) else g(x);`

Many of MiniCalc's built-in functions use the pass-by-expression mechanism. For example, `@IfNull` takes two parameters and always evaluates the first, but only evaluates the second if the first is null. You could define your own `IfNull` function using pass-by-expression:

```
function IfNull(Value, ?Default);
begin
  if @IsNull(Value) then
    @Evaluate(Default)
  else
```

```
    Value;  
end;
```

1.7 Nested Blocks

Nested Blocks

It is sometimes helpful to define temporary variables to use when computing a complex expression. The location for storing new variables depends on the current environment used to evaluate an expression. Sometimes there is no environment, in which case the Global Scope will hold new variables. If an environment does exist, new variables will be stored there. However, some environments cannot accept new variables: for example, an environment implemented as a wrapper around a smart record. Even if it could be guaranteed that a new variable would be created, it would still be desirable to create temporary variables in a nested scope, so that they would not "pollute" the environment.

MiniCalc supports local variables with the **var**, **begin**, and **end** keywords. Here is an example:

```
var  
  x = r*(r-1);  
begin  
  (x+2) / (x+1);  
end;
```

The use of the temporary, *x*, makes the above easier to read than this:

```
(r*(r-1) + 2) / (r*(r-1) + 1)
```

You can define multiple variables in the **var** clause, and you can supply multiple expressions in the **begin/end** block:

```
var  
  pi2 = @Sqr(@Pi),  
  z = (r+4)/pi2;  
begin  
  x := @Sin(z);  
  y := @Cos(z);  
end;
```

The value of the overall **var/begin/end** statement is the value of the last expression in the **begin/end** block. For example,

```
var  
  temp;  
begin  
  temp := x;  
  x := y;  
  y := temp;  
  x*y;  
end;
```

The above code swaps the value of *x* and *y*, then returns their product. This example also illustrates that the initialization expression is optional when declaring a variable.

The semi-colon before the **begin** keyword is optional, as is the one after the **end** keyword. For example:

```
var  
  x = r*(r-1)  
begin  
  r := (x+2) / (x+1)  
end  
  
2*r
```

Using Variables in a Nested Block

Within a nested block, the following rules apply when accessing variables:

- Variables defined in the nested block hide variables from the enclosing environment.
- The assignment operator will not define a new variable. The variable must be defined in the nested block, or exist in the enclosing environment. Otherwise, an exception will be raised.
- When the nested block terminates, the associated variables are discarded.

1.8 Control Structures

Control Structures

Although MiniCalc is primarily intended to support simple expressions, it does offer several structured programming constructs, including if/then/else statements, for loops (Delphi style and C/C++/C#/Java style), for-in loops, and while loops.

If Statements

MiniCalc's if/then/else statement works just like it does in Delphi.

While Statements

MiniCalc's while statement works just like it does in Delphi.

For Loops: Delphi-Style

MiniCalc offers a for loop that works just like Delphi's for loop.

For Loops: Java-Style

MiniCalc offers a for loop that works just like the for loop in C/C++/C#/Java.

For-In Loops

MiniCalc offers for-in loop (as in Delphi 9 and above). Here is an example:

```
for Item in List do
  Process(Item);
```

The collection over which the for-in loop iterates may be an *IVector*, an array, a string, or an *IAssociation* (see the *@AsVect* for details on how this works). Note that the for-in statement makes a copy of the collection **unless** the collection is an *IVector*, *IMap*, or *IIntfMap*. In these cases, an iterator will be used to avoid copying the container. You should avoid modifying the container (adding or removing elements) in the body of the for-in loop.

1.9 Objects

Objects

The full power of MiniCalc is realized when combined with objects that support various interfaces known to MiniCalc. There are two ways you can use these objects in MiniCalc.

The MiniCalc dot operator requires an *IAssociation* interface. Examples of such objects include *IStringList* (when using the name/value pairs feature), *IMap* and *IIntfMap*, *IDataSet*, and *IPropertyBag*. When a variable refers to one of these objects, you can use the dot operator to access the object's fields. For example, the code below shows how to add a business object to a MiniCalc environment, then access the fields of the object from within MiniCalc:

```
procedure RunMain;
var
  Trade: ITrade;
  Env: ICalcEnv;
  Expr: ICalcExpr;
  V: Variant;
begin
  Trade := NewTrade;
  Trade.DollrAmt := -45.88;
  Env := NewCalcEnv(csIgnoreCase);
  Env['Trd'] := Trade;
  Expr := ParseCalcExpr('@Abs(Trd.DollrAmt)');
  V := Expr.Evaluate(Env);
```

```
Writeln(V);
end;
```

To access fields names with spaces or other special characters, simply quote the field name. For example:

```
Expr := ParseCalcExpr( '@Abs(Trd."Dollar Amount")' );
```

The other way to use objects in MiniCalc is with various functions. For example, the *@Length* function returns the length of a string, or the number of elements in a variant array or *IVector* object. The *@Aggregate* function computes an aggregate expression over all elements of a variant array, *IVector*, or *Iterator*. Other examples may be found in the section on functions.

1.10 Built-In Constants

Built-In Constants

The following table describes the built-in constants which are automatically defined in the global environment.

Name	Type	Notes
@True	Boolean	True constant (deprecated; use "true" keyword instead)
@False	Boolean	False constant (deprecated; use "false" keyword instead)
@Nil	Interface	nil Interface pointer
@Null	Null	Null Variant
@Pi	Double	pi
@e	Double	e (the base of natural logarithms)
@NaN	Double	NaN (floating-point not-a-number)
@Infinity	Double	Infinity (floating-point infinity)
@NegInfinity	Double	-Infinity (floating-point negative infinity)
@MaxDouble	Double	The maximum value of the double precision floating-point type
@MinDouble	Double	The minimum value of the double precision floating-point type
@MaxInt	Int/Int64	The largest variant-compatible integer value: High(Integer) on Delphi 5; High(Int64) on Delphi 6 and later
@MinInt	Int/Int64	The smallest variant-compatible integer value: Low(Integer) on Delphi 5; Low(Int64) on Delphi 6 and later
@csCaseSensitive	Integer	Ord(csCaseSensitive)
@csIgnoreCase	Integer	Ord(csIgnoreCase)
@dupAllow	Integer	Ord(dupAllow)
@dupError	Integer	Ord(dupError)
@dupIgnore	Integer	Ord(dupIgnore)
@rfReplaceAll	Integer	Ord(rfReplaceAll)
@rfIgnoreCase	Integer	Ord(rfIgnoreCase)
@tsSingleThread	Integer	Ord(tsSingleThread)

@tsThreadSafe	Integer	Ord(tsThreadSafe)
@varEmpty	Integer	The varXxx constants correspond to the constants of the same name defined in the <i>System</i> unit. They are used in conjunction with <i>@VarType</i> and <i>@ArrayOfType</i> .
@varNull	Integer	See note for varEmpty.
@varSmallint	Integer	See note for varEmpty.
@varInteger	Integer	See note for varEmpty.
@varSingle	Integer	See note for varEmpty.
@varDouble	Integer	See note for varEmpty.
@varCurrency	Integer	See note for varEmpty.
@varDate	Integer	See note for varEmpty.
@varOleStr	Integer	See note for varEmpty.
@varDispatch	Integer	See note for varEmpty.
@varError	Integer	See note for varEmpty.
@varBoolean	Integer	See note for varEmpty.
@varVariant	Integer	See note for varEmpty.
@varUnknown	Integer	See note for varEmpty.
@varShortInt	Integer	See note for varEmpty.
@varByte	Integer	See note for varEmpty.
@varWord	Integer	See note for varEmpty.
@varLongWord	Integer	See note for varEmpty.
@varInt64	Integer	See note for varEmpty.
@varStrArg	Integer	See note for varEmpty.
@varString	Integer	See note for varEmpty.
@varAny	Integer	See note for varEmpty.

1.11 Filters, Comparators, and Transforms

Filters, Comparators, and Transforms

MiniCalc extends *DObject.pas*, by offering filters, comparators, and transforms that are defined by MiniCalc expressions. These objects are useful in conjunction with *IList* (defined in *DSLList.pas*) and *Iterator* (defined in *DGenUtil.pas*) and related types.

VariantFilterExpr

This function creates a new *IFilter* object, defined by a MiniCalc expression:

```
function VariantFilterExpr(
  const MiniCalcExpr: string): IFilter; overload;
```

```
function VariantFilterExpr(  
    const MiniCalcExpr: ICalcExpr): IFilter; overload;
```

Within the MiniCalc expression, the filter value is made available via the variable "x". One use of IFilter is the FindMany method of IList:

```
procedure Demo(const List: IList);  
var  
    LargeItems: IList;  
begin  
    LargeItems := List.FindMany(VariantFilterExpr('x > 100'));  
end;
```

The above example creates a second list, LargeItems, that contains only items from the original list that are greater than 100.

VariantCompareExpr

This function creates a new ICompare object, defined by a MiniCalc expression:

```
function VariantCompareExpr(  
    const MiniCalcExpr: string): ICompare; overload;  
  
function VariantCompareExpr(  
    const MiniCalcExpr: ICalcExpr): ICompare; overload;
```

Within the MiniCalc expression, the values to compare are made available via the variables "x" and "y". One use of ICompare is the Sort method of IList:

```
procedure Demo(const List: IList);  
begin  
    List.Sort(VariantCompareExpr('@CompareC(@Abs(x), @Abs(y))'));  
end;
```

The above example sorts items in the list by magnitude (absolute value).

VariantTransformExpr

This function creates a new ITransform object, defined by a MiniCalc expression:

```
function VariantTransformExpr(  
    const MiniCalcExpr: string): ITransform; overload;  
  
function VariantTransformExpr(  
    const MiniCalcExpr: ICalcExpr): ITransform; overload;
```

Within the MiniCalc expression, the value to transform is made available via the variable "x". One use of ITransform is the iterator wrapper:

```
procedure Demo(const List: IList);  
var  
    i: IIterator;  
    UpperList: IList;  
begin  
    i := NewIterator(List);  
    i := NewIterator(i, VariantTransformExpr('@UpperCase(x)'));  
    UpperList := NewList(i, csIgnoreCase);  
end;
```

The above example creates an iterator that returns the items in List, converting each item to uppercase along the way. This iterator is used to populate a second list (UpperList).

Index

@

@Abs

Built-In Functions 4

Filters, Comparators, and Transforms 18

Objects 16

@Acos 4

@Acosh 4

@Acot 4

@Acoth 4

@Acsc 4

@Acsch 4

@Add 4

@AddPaths 4

@Aggregate

Aggregate Functions 11

Built-In Functions 4

Objects 16

@ArrayOf 4

@ArrayOfType 4

@Asec 4

@Asech 4

@Asin 4

@Asinh

@Atan 4

@AsVect 4

@Atanh 4

@Avg

Aggregate Functions 11

Built-In Functions 4

@Break 4

@Cast 4

@Ceil 4

@Ceil64 4

@Chr

@CompareC 4

@Collection 11

@CompareC 18

@CompareI 4

@Continue 4

@Copy 4

@Cos 4

@Cosh 4

@Cot 4

@Coth 4

@Count

Aggregate Functions 11

Built-In Functions 4

@Csc 4

@Csch 4

@Date 4

@Defined 4

@DSSimpleRoundTo 4

@Equal 4

@Error 4

@Evaluate 4

@Exit 4

@Exp 4

@Expr 4

@ExtractFileBase 4

@ExtractFileExt 4

@ExtractFileName 4

@ExtractFilePath 4

@FileExists 4

@FindFirst 4

@Floor 4

@Floor64 4

@For 4

@FormatDateTime 4

@Frac 4

@GetElem 4

@GetKeys 4

@GetNamePart 4

@GetValue 4

@GetValuePart 4

@GetValues 4

@GlobalEnv 4

@HasValue 4

@High 4

@Hypot

@Int 4

@IfBlank 4

@IfNull

@InC 4

@IndexOf 4	@NewFileInStream 4
@Inl 4	@NewFileOutStream 4
@Inspect 4	@NewIntfList 4
@Int64 4	@NewIntfMap 4
@IsArray 4	@NewList
@IsAssoc 4	@NewMap 4
@IsBlank 4	@NewPersist 4
@IsExpr 4	@NewPropBag 4
@IsFunc 4	@NewStringList 4
@IsInfinite 4	@NiceDate 4
@IsIterator 4	@NiceDateTime 4
@IsList 4	@NotDefined 4
@IsMap 4	@NotEqual 4
@IsNaN 4	@NotNull 4
@IsNull 4	@Now 4
@IsObject 4	@Null 2
@IsPersist 4	@Parse 4
@IsPrime 4	@ParseCSV 4
@IsVect 4	@Pos 4
@IsZero 4	@PrintPropBag 4
@Item 11	@Raise 4
@Length	@Return 4
Built-In Functions 4	@Reverse 4
Objects 16	@Root 4
@Ln 4	@Round 4
@LocalEnv 4	@RoundToDigits 4
@Log10 4	@SameObject 4
@Log2 4	@SameText 4
@LogN	@Sec
@NextPrime 4	@Sech 4
@Low 4	@SetElem 4
@LowerCase	@SetGlobal
@LoadIniFile 4	Built-In Functions 4
@Max	Environments and Variables 3
Aggregate Functions 11	@SetValue
Built-In Functions 4	@Sign 4
@MaxAgg 4	@SimpleRoundTo 4
@Min	@Sin 4
Aggregate Functions 11	@Sinh 4
Built-In Functions 4	@Sqr 4
@MinAgg 4	@Sqrt 4
@NewBufFileInStream 4	@StringReplace 4
@NewBufInStream 4	@Sum

Aggregate Functions 11

Built-In Functions 4

@Supports 4

@Tan 4

@Tanh 4

@Time 4

@Trim 4

@TrimLeft 4

@TrimRight 4

@Unassigned

@UpperCase 4

@UpperCase 18

@VarType 4

@While 4

A

Aggregate Functions 11

AggregateArray 11

AggregateIterator 11

AggregateVector 11

AsBool 1

AsString 1

B

Built-In Constants 17

Built-In Functions

Aggregate Functions 11

Built-In Functions 4

C

Calling Functions 4

CheckCount 12

Comments 2

Comparators 18

Constants 17

Control Structures 16

D

Defining New Functions 11

Defining New Functions in Delphi 12

Defining New Functions in MiniCalc 13

Dot Operator 16

E

Environments 3

Environments and Variables 3

Evaluate

Environments and Variables 3

Example 1

Objects 16

Example 1

F

false 2

Filters 18

Filters, Comparators, and Transforms 18

Functions 4

G

Global Environment

Aggregate Functions 11

Built-In Functions 4

Built-In Constants 17

Environments and Variables 3

I

ICalcEnv

Environments and Variables 3

Objects 16

ICalcExpr

Defining New Functions in Delphi 12

Defining New Functions in MiniCalc 13

Environments and Variables 3

Example 1

Objects 16

ICalcExprList 11

ICalcFunc

Aggregate Functions 11

Defining New Functions in Delphi 12

Defining New Functions in MiniCalc 13

Iterator 11

IVector 11

M

MiniCalc 1

MiniCalcGlobals 3

N

Nested Blocks 15

NewCalcEnv 3

NewCalcExpr 16

NewCalcFunc

 Aggregate Functions 11

 Defining New Functions in Delphi 12

O

Objects 16

Operands 2

Operators 2

Operators and Operands 2

Overview 1

P

ParseCalcExpr

 Environments and Variables 3

 Example 1

 Objects 16

Precedence 2

T

Transforms 18

true 2

U

User-Defined Functions

 Defining New Functions in Delphi 12

 Defining New Functions in MiniCalc 13

V

Variables 3

VariantCompareExpr 18

VariantFilterExpr 18

VariantTransformExpr 18

