Documentation for TFormulaParser

1. How to instanciate the formula parser

```
Var
  FP: TFormulaParser;
Begin
  FP := TFormulaParser.Create;
  ...
```

That's it!

2. How to use it

First of all, you need a string that representates a formula. For example: '4+5'
This string is passed to the formula parser through its property Formula.

```
  ...
  FP.Formel := '4+5';
  ...
```

Now you can get the result by calling the Calculate function. Which is defined as followed:

function Calculate(var Answer, Error: String): Boolean;

The function returns true if it was able to calculate the formula. The formula's result is given in the answer string and error is of no interest. If the function returns false, it was unable to calculate the formula; error contains an error-string and answer is of no interest.

Example:

```
Var
  FP: TFormulaParser;
  Answer, Error: String ;
  B: Boolean;
Begin
  FP := TFormulaParser.Create;
  FP.Formula := '4+5';
  B := FP.Calculate(Answer, Error);
  // The variables will now contain the following content:
  // B = true;   Answer = '9';   Error is undefined
  FP.Formula := '4+5+';
  B := FP.Calculate(Answer, Error);
  // The variables will now contain the following content:
  // B = false;   Answer is undefined;
  // Error = 'Syntax error. More arguments expected.'
  ...
```

3. How to use variables

To use variables it is necessary to tell the parser what variables are known and what value they have. Therefore the parser calls a callback-routine whenever it finds an unknown value which cannot be anything else than a variable.

The callback-routine is stored in the Callback property and is of
type
TGetVariableProc = function(Name: String; var Answer, Error: String): Boolean of object;.

The function gets a string Name which contains the variables name. Its result must be true if the variable was successfully found and Answer must be the string containing the variables value. If the function is unable to find the desired variable it must return false and set Error to an error-string.

Example:

```
Function TForm1.OnVariable(Name : String; var Answer, Error:
  String): Boolean;
Begin
  If Name = 'X' Then Begin
    Answer := '5';
    Result := true;
  End Else Begin
    Error := 'unknown variable';
    Result := false;
  End;
End;

Procedure TForm1.Calculate
Var
  FP: TFormulaParser;
  Answer, Error: String ;
  B: Boolean;
Begin
  FP := TFormulaParser.Create;
  FP.CallBack := OnVariable;
  FP.Formula := '4+X';
  B := FP.Calculate(Answer, Error);
  // The variables will now contain the following content:
  // B = true;   Answer = '9';   Error is undefined
  FP.Formula := '4+Y';
  B := FP.Calculate(Answer, Error);
  // The variables will now contain the following content:
  // B = false;   Answer is undefined;
  // Error = 'unknown variable'
  ...
```

4. Creating a new Set of Operators

TFormulaParser contains a List of Operators. By default these are the binary Operators ‘+’, ‘-‘, ‘*’, ‘/’ and the unary Operators ‘-‘, ‘sin’, ‘cos’, ‘tan’, ‘arcsin’, arccos’ and ‘arctan’.
To remove any of these Operators one must call the RemoveUnaryOperatorByName or RemoveBinaryOperatorByName methods of TFormulaParser.
Example:

```
  ...
  FP.RemoveBinaryOperatorByName('+');
  ...
```

It is also possible to put an operator back into TFormulaParser. This is done by instanciating an operator and passing it to the AddOperator method of TFormulaParser.
Example:

```
...
var
  OP: TFormelOperator;
  FP: TFormelParser;
begin
  FP := TFormelParser.Create;
  FP.RemoveBinaryOperatorByName('+');
  OP := TFormulaOperator_Addition.create(1);
  FP.AddOperator(OP);
  ...
```

In this example the ‘+’-operator was removed and then put back in. The number behind the constructor specifies the priority of the operator. In the example the ‘+’ operators priority was raised from 0 to 1, which means, that it binds stronger than the binary ‘-‘ and equally strong as ‘*’ and ‘/’.
Resetting the priority makes only sense, when non-standard operators are inserted, whos priorities are between two or more existing operators. The higher priorities must be increased then. Furthermore in most cases it makes no sense to create unary operators that have lower priorities than binary. Be careful with this, because a binary operator that is followed directly by a unary of lower priority will cause a syntax-error.

Here’s a List of the standard operators their classes and standard priorities and parameter:

```
'+'       TFormelOperator_Addition.create         0    binary
'-'       TFormelOperator_Subtraction.create      0    binary
'*'       TFormelOperator_Multiplication.create   1    binary
'/'       TFormelOperator_Division.create         1    binary
'-'       TFormelOperator_UnaryMinus.create       2    unary
'sin'     TFormelOperator_Sinus.create            2    unary
'cos'     TFormelOperator_Cosinus.create          2    unary
'tan'     TFormelOperator_Tangens.create          2    unary
'arcsin'  TFormelOperator_ArcusSinus.create       2    unary
'arccos'  TFormelOperator_ArcusCosinus.create     2    unary
'arctan'  TFormelOperator_ArcusTangens.create     2    unary
```

5. Adding operators

Operators are derived from the basic operators class TFormulaOperator which is defined as:

```
type TFormulaOperator = class
  private
    F_Name: String;
    F_Unary: Boolean;
    F_Priority: Integer;
  public
    function Calculate(Argument1, Argument2: String; var
      Answer, Error: String):Boolean; virtual; abstract;
    property Name: String read F_Name;
    property Unary: Boolean read F_Unary;
    property Priority: Integer read F_Priority;
  end;
```

To create a new operater, you have to override the calculate function and define a constructor.
Example:

```
type TFormulaOperator_Addition = class(TFormulaOperator)
  public
    function Calculate(Argument1, Argument2: String; var
      Answer, Error: String):Boolean; override;
    constructor Create(Priority: Integer);
  end;
```

The constructor does not need to be in a special cast, but it is useful to pass the priority through this way.
Example:

```
constructor TFormelOperator_Addition.Create(Priority:
  Integer);
begin
  F_Name := '+';
  F_unary := false;
  F_Priority := Priority;
end;
```

When called from the parser the Calculate function is given two strings: Argument1 and Argument2. If the operator is binary, these two strings contain the operands which the operator shall work on. If the operator is unary, then only Argument1 is set but Argument2 is undefined. The Calculate function shall then work on the operand(s) and store the result in Answer. If this was successful, then Result must be true. If not, then Result must be false and an error-string must be stored in Error.

Example:

```
function TFormulaOperator_Addition.Calculate(Argument1,
  Argument2: String; var Answer, Error: String): Boolean;
var
  x,k: Extended;
begin
  result := true;
  Answer := 'ERROR';
  if not TextToFloat(PChar(Argument1), k, fvExtended) then
begin
    Error := '"' + Argument1 + '" is not a valid floating
number. Addition not possible.';
    result := false;
    exit;
  end;
  if not TextToFloat(PChar(Argument2), k, fvExtended) then
begin
    Error := '"' + Argument2 + '" is not a valid floating
number. Addition not possible.';
    result := false;
    exit;
  end;
  Answer := FloatToStr(x + k);
end;
```

A new operator can then be activated by passing it to AddOperator method of TformulaParser, like described in part 4.

6. Extra features

To change the parser's operators it also possible to derive a new class from TFormelParser and to override the CreateOperators procedure. Operators that are of type as above have to be instanciated and added in the F_FormelOperatorList, which is of type TList. This is the job of the CreateOperators procedure. It is important that the operators are inserted sorted by priority, or the parser wont be able to find them.
Below is the code of the default CreateOperators method.

```
procedure TFormelParser.CreateOperators;
var
  AOperator: TFormelOperator;
begin
  AOperator:= TFormelOperator_Addition.create(0);
  F_FormelOperatorList.AddOperator(AOperator);
  AOperator:= TFormelOperator_Subtraction.create(0);
  F_FormelOperatorList.AddOperator(AOperator);
  AOperator:= TFormelOperator_Multiplication.create(1);
  F_FormelOperatorList.AddOperator(AOperator);
  AOperator:= TFormelOperator_Division.create(1);
  F_FormelOperatorList.AddOperator(AOperator);
```

```
  AOperator:= TFormelOperator_UnaryMinus.create(2);
  F_FormelOperatorList.AddOperator(AOperator);
  AOperator:= TFormelOperator_Sinus.create(2);
  F_FormelOperatorList.AddOperator(AOperator);
  AOperator:= TFormelOperator_Cosinus.create(2);
  F_FormelOperatorList.AddOperator(AOperator);
  AOperator:= TFormelOperator_Tangens.create(2);
  F_FormelOperatorList.AddOperator(AOperator);
  AOperator:= TFormelOperator_ArcusSinus.create(2);
  F_FormelOperatorList.AddOperator(AOperator);
  AOperator:= TFormelOperator_ArcusCosinus.create(2);
  F_FormelOperatorList.AddOperator(AOperator);
  AOperator:= TFormelOperator_ArcusTangens.create(2);
  F_FormelOperatorList.AddOperator(AOperator);
end;
```

TFormulaParser contains some other methods and properties that might be useful. Here's a list:

```
TFormulaParser = class
protected
  F_FormulaOperatorList: TFormulaOperatorList;
```
This is the Object that stores all the Operators. It is described above.
```
  procedure CreateOperators; virtual;
```
This is described above.
```
public
  constructor Create;
```
This is described in part 1.
```
  destructor Destroy; override;
```
Simple destructor, nothing special;
```
  procedure AddOperator(Operator: TFormulaOperator);
```
Adds an Operator to F_FormulaOperatorList. The Operator will automaticly be placed at the right position. More Info in part 4.
```
  procedure RemoveUnaryOperatorByName(Name: String);
```
This is described in part 4.
```
  procedure RemoveBinaryOperatorByName(Name: String);
```
This is described in part 4.
```
  procedure RemoveAllOperators;
```
Removes all Operators from the F_FormulaOperatorList..
```
  function GetOperatorCount: Integer;
```
Returns the amount of Operators stored in F_FormulaOperatorList.
```
  function GetHighestPriority: Integer;
```
Returns the highest priority of operands stored in F_FormulaOperatorList.
```
  procedure ReorganizePriorities;
```
Reorganizes the F_FormulaOperatorList. After this no priority will be skipped in the F_FormulaOperatorList. If needed some priorities are lowered to claim the desired order. The Operators will also be sorted by priority.
```
  function Parse(var Error: String): Boolean;
```
Parses the Formula without calculating the result. Only the parse-tree will be created an the syntax is checked, but the callback is not used. If the syntax was correct, true will be returned, if not then false is the result and Error is set.

The Tree remains as long as the formula is not changed. The Calculate method will use the tree.

```
function Calculate(var Answer, Error: String): Boolean;
```

Calculates the result. If no parse-tree was created this will be done previously. If it is, then the calculations speed is increased. Running the same formula while changing the variable-callback is much more time-efficient than creating a new formula with other constant values.

More Info in part 2.

```
property Formula: String read getFormula write setFormula;
```

Described in 2.

```
property Callback: TGetVariableProc read F_Callback write
      F_Callback;
```

Decribed in 3.

```
end;
```

7 Further information

For knew information about this and other components, plase have at look at:

http://www.ibhalbauer.de/Components/components.html

Formula Parser is deliverd 'as is'. The author does not guarantee for anything. Use on own risk.

If any questions are left open or you have some ideas, please contact:

info@ibhalbauer.de

Thanks
Ralph Halbauer