Parser10 unit

The PARSER10 unit, together with the P10BUILD unit contain a fully featured mathematical expression parser for Delphi 1, Delphi 2, Delphi 3 and C++ Builder 1.0.

TParser parses and evaluates mathematical expressions specified at runtime. Its performance is remarkable only 40-80% slower than similar compiled expression and it is by far the fastest parser on the freeware market.

If you do not have the time to read through all of this help file, the QuickStart section may be interesting.

Components
TCustomParser
TParser

Exceptions
EMathParserError
EBadName
EExpressionHasBlanks
EExpressionTooComplex
EMissMatchingBracket
EParserInternalError
ESyntaxError
ETooManyNestings

Types
ParserFloat
POperation
PParserFloat
TMathProcedure
TOperation
TParserExceptionEvent
TToken

QuickStart

The programming interface for TParser is simple:

specify values for predefined variables in properties
A,B,C,D,E,X,Y or T;
optionally add new variables or removing existing ones;
specify expression to be evaluated in Expression property;
optionally change value of variables
retrieve computed value in Value property.

Example

```
Parser1.X := 100;
Parser1.Y := 200;
Parser1.Variable['z'] := 20;
Parser1.Expression := 'sin(x)*cos(y)+z';
Result := Parser1.Value;
```

If you want to compute several values of the same expression with different sets of variables, specify the expression once **only** as this operation is rather time consuming, then assign new values for variables and
retrieve the Value property as many times as you wish.

Example

```
for i := 1 to 100 do
begin
  Parser1.X := i;
  Result := Parser1.Value;
end;
```

You also may add your own variables by specifying their names and values using the SetVariable method, accessing them via a property or through direct memory access (as usual variable names are not case-sensitive).

Example (the following lines are all equivalent)

```
Parser1.Variable['Test'] := 100;
Test := Parser1.Variable['Test'];

Parser1.SetVariable('Test', 100);
Test := Parser1.GetVariable('Test');
```

```
var
  PTest : PParserFloat; { pointer to memory }
begin
  PTest := Parser1.SetVariable('Test', 100);
  PTest^ := 200; { set 'Test' = 200 }
  Test := PTest^;
```

There is no limit for expression length in the 32bit version, while the 16bit version has a restriction of 255 characters.

Predefined constants:
PI

Accepted operators: + , - , * , / , ^ , MOD, DIV
[MOD and DIV implicitly perform a trunc() on their operands]

The following functions are supported; it doesn't matter if you use lower or upper case:

[NOTE: to activate some additional functions you need to remove the `SpeedCompare` conditional define in PARSER10.PAS !]

COS, SIN, SINH, COSH, TAN, COTAN, ARCTAN, ARG,

EXP, LN, LOG10, LOG2, LOGN,

SQRT, SQR, POWER, INTPOWER,

MIN, MAX, ABS, TRUNC, INT, CEIL, FLOOR,

HEAV (heav(x) is =1 for x>0 and =0 for x<=0),
SIGN (sign(x) is 1 for x>1, 0 for x=0, -1 for x<0),
ZERO (zero(x) is 0 for x=0, 1 for x<>0),
PH (ph(x) = x - 2*pi*round(x/2/pi))
RND (rnd(x) = int(x) * Random)
RANDOM (random(X) = Random; the argument X is not used)

Adding your own functions is easy, too. Either use…

`AddFunctionOneParam` or
`AddFunctionTwoParam`

… if you do not want to create a new class, or create a new class, inheriting from `TParser` and add your functions to the lists, as demonstrated in the source code of `TParser.Create` in `PARSER10.PAS`.

Important:
Do not use blanks (#32) in the expression. The parser is unable to handle these and raises an exception in response.

You can use bracketing (nestings) up to a level of 20. This can be increased at the expense of stack consumption by changing the line

```
maxBracketLevels = 20;
```

in `P10BUILD.PAS`. This should not be a problem.

If you get an "Expression too complex" exception increase

```
maxLevelWidth = 50;
```

in `P10BUILD.PAS`. This should never happen.

You can define your *own* variables at *runtime*, in code you will use

```
Parser1.Variable.Add('NAME', 123456)
```

or simply (but slow)

```
Parser1.Variable['ANOTHER'] := 1.23
```

See the demonstration program for better techniques.

Important:
The used *mathematical* routines behave exactly like Delphi runtime code in case of errors. Some people feel that this is a problem.

This is not a parsing issue, but rather a lack of attention to the actual maths part (which is in a few places is sloppy - deliberately)... Most probably you will be using your own mathematical routines which are faster, more reliable, and provide more functionality.

EMathParserError Exception

Unit
Parser10

Declaration
```
EMathParserError = class(Exception);
```

Description
This is the base exception for all exceptions raised by the parser.

ESyntaxError Exception

Unit
Parser10

Declaration
```
ESyntaxError = class(EMathParserError);
```

Description
In case of an expression syntax error this exception will be raised.

EExpressionHasBlanks Exception

Unit
Parser10

Declaration
```
EExpressionHasBlanks = class(EMathParserError);
```

Description
`TCustomParser` and `TParser` cannot handle expressions that contain blanks.

If the parsing engine nevertheless gets passed an expression that contains blanks, it will first attempt to remove trailing and leading blanks. Only if after this blanks are left in the expression this exception will be raised.

EExpressionTooComplex Exception

Unit
Parser10

Declaration
```
EExpressionTooComplex = class(EMathParserError);
```

Description
If the expression assigned to the `Expression` property is too complex, this exception will be raised; if the dynamic expression matrix code has been activated this exception will never occur. In case the static expression matrix code is used it is highly unlikely that this exception will ever occur (the limits have been set very generously).

ETooManyNestings Exception

Unit
Parser10

Declaration
```
ETooManyNestings = class(EMathParserError);
```

Description
If compiled with the static expression matrix code, this exception will be raised, when the number of bracketing levels exceeds the number defined in `P10BUILD.PAS` by `maxBracketLevels` (by default set to 20).

There is no limit in the number of bracketing levels used if the component has been compiled with the dynamic expression matrix code.

EMissMatchingBracket Exception

Unit
Parser10

Declaration
```
EMissMatchingBracket = class(EMathParserError);
```

Description
If a (opening or closing) bracket is missing from the expression, this expression will be raised, indicating the number and type of missing brackets.

EBadName Exception

Unit
Parser10

Declaration
```
EBadName = class(EMathParserError);
```

Description
This exception will be raised each time a variable or function is added to the parser that does not match the Pascal naming convention. Additionally, functions or variables containing the character sequences "MOD" and "DIV" are not allowed, as the parser would not be able to discriminate between these operators and the variable / function in an expression.

EParserInternalError Exception

Unit
Parser10

Declaration
```
EParserInternalError = class(EMathParserError);
```

Description
You hopefully we will never see this exception, as it indicates that some internal assumptions went wrong.

If you get this exception, please contact **Stefan.Hoffmeister@poboxes.com** with the expression and the exact circumstances that caused the problem.

TCustomParser Component

Properties Methods Events

Unit
Parser10

Description
This is the base class of the parser, defining all of the functionality. Its only child in the `Parser10` unit, `TParser`, only adds some access methods to pre-defined variables and supplies some instantly known functions to the parsing and evaluation engine.

Properties
Run-time only
Key properties

A	PascalNumberformat	
B	T	
C	Value	
D	Variable	
E	X	
Expression	Y	
LinkedOperationList	ParserError	

Methods
Key methods

AddFunctionOneParam	ClearVariable	ParseExpression
AddFunctionTwoParam	ClearVariables	SetVariable
ClearFunction	FreeExpression	VariableExists
ClearFunctions	GetVariable	

Events
Key events
OnParserError

A property

Applies to
TCustomParser, TParser

Declaration
```
property A: ParserFloat;
```

Description
This is a pre-defined variable that is accessible at design-time. Its only purpose is its presence at design-time; apart from that it is a normal variable.

B property

Applies to
TCustomParser, TParser

Declaration
```
property B: ParserFloat;
```

Description
This is a pre-defined variable that is accessible at design-time. Its only purpose is its presence at design-time; apart from that it is a normal variable.

C property

Applies to
TCustomParser, TParser

Declaration
```
property C: ParserFloat;
```

Description
This is a pre-defined variable that is accessible at design-time. Its only purpose is its presence at design-time; apart from that it is a normal variable.

D property

Applies to
TCustomParser, TParser

Declaration
```
property D: ParserFloat;
```

Description
This is a pre-defined variable that is accessible at design-time. Its only purpose is its presence at design-time; apart from that it is a normal variable.

E property

Applies to
TCustomParser, TParser

Declaration
```
property E: ParserFloat;
```

Description
This is a pre-defined variable that is accessible at design-time. Its only purpose is its presence at design-time; apart from that it is a normal variable.

T property

Applies to
TCustomParser, TParser

Declaration
```
property T: ParserFloat;
```

Description
This is a pre-defined variable that is accessible at design-time. Its only purpose is its presence at design-time; apart from that it is a normal variable.

X property

Applies to
TCustomParser, TParser

Declaration
```
property X: ParserFloat;
```

Description
This is a pre-defined variable that is accessible at design-time. Its only purpose is its presence at design-time; apart from that it is a normal variable.

Y property

Applies to
TCustomParser, TParser

Declaration
```
property Y: ParserFloat;
```

Description
This is a pre-defined variable that is accessible at design-time. Its only purpose is its presence at design-time; apart from that it is a normal variable.

ParserError property

Applies to
TCustomParser, TParser

Declaration
```
property ParserError: boolean;
```

Description
This property will be set to True if the least recently passed expression (using either `ParseExpression` or the `Expression` property) was parsed without error. It will be False otherwise.

LinkedOperationList property

Applies to
TCustomParser, TParser

Declaration
```
property LinkedOperationList: POperation;
```

Description
The `LinkedOperationList` property points to the first element of a sequence of mathematical operations.

Please do never modify this property or the pointer therein unless you are aware of all the side-effects of this action.

Variable property

Applies to
TCustomParser, TParser

Declaration
```
property Variable[const VarName: string]: extended;
```

Description
Accessing the `Variable` property is yet another way to get and set a variable's value. This is the by far slowest way to access variable values. It is only provided for completeness and to allow more elegant code.

It is strongly recommended that you use the methods `GetVariable` and `SetVariable` directly.

Value property

Applies to
TCustomParser, TParser

Declaration
```
property Value: extended;
```

Description
To evaluate an expression simply query this property.

Example
```
Parser1.Variable['theta'] := 2.5;
Parser1.Expression := '100+100+sin(theta)';
Result := Parser1.Value;
```

Expression property

Applies to
TCustomParser, TParser

Declaration
```
property Expression: string;
```

Description
Setting the Expression property will automatically parse the expression.

If an error occurs while the expression is parsed, an exception may be raised. If `OnParserError` is not assigned the exception will fall through to the next exception handler. Otherwise the `OnParserError` event will be triggered.

Example
```
Parser1.Variable['theta'] := 2.5;
Parser1.Expression := '100+100+sin(theta)';
Result := Parser1.Value;
```

PascalNumberformat property

Applies to
TCustomParser, TParser

Declaration
```
property PascalNumberformat: boolean;
```

Description
This property determines the expected format of numbers in an expression.

If set to True the standard Pascal number format with the '.' being the decimal separator and no thousand separator will be used.

If set to False, numbers adhering to the current international country settings will be expected. `DecimalSeparator` and `ThousandSeparator` as declared in the `SysUtils` unit will be used.

ParseExpression method

Applies To
TCustomParser, TParser

Declaration
```
function ParseExpression(const AnExpression: string): boolean;
```

Description
Use `ParseExpression` to translate an expression into the internal representation of the component.

Calling this function will automatically assign the passed argument to the `Expression` property.

The function will return False, if an error occurred while parsing the expression; True otherwise.

FreeExpression method

Applies To
TCustomParser, TParser

Declaration
```
procedure FreeExpression;
```

Description
The `FreeExpression` method will discard the internal structures built by `ParseExpression`.

SetVariable method

Applies To
TCustomParser, TParser

Declaration
```
function SetVariable( VarName: string;
                      const Value: extended): PParserFloat;
```

Description
Use this method to assign a value to a memory. If the variable does not exist, it will be created.

The `PParserFloat` returned points to the place in memory where the variable actually is stored; to speed up assignment you can directly assign data to the memory area.

Example
```
var
  APParserFloat: PParserFloat;
begin
  APParserFloat := Parser1.SetVariable('Test', 1.333);
  ShowMessage(FloatToStr(APParserFloat^));
```

GetVariable method

Applies To
TCustomParser, TParser

Declaration
```
function GetVariable(const VarName: string): extended;
```

Description
Call `GetVariable` with the name of the variable to retrieve the variable's current value. If the variable does not exist, 0.0 is returned.

If you happen to know the location of the variable in memory, for instance because you stored the pointer returned by `SetVariable`, it is much faster to read the memory directly.

AddFunctionOneParam method

Applies To
TCustomParser, TParser

Declaration
```
procedure AddFunctionOneParam( const AFunctionName: string;
                               const Func: TMathProcedure);
```

Description
This procedure will add under the name of `AFunctionName` the code pointed to by `Func`.

The passed function code is expected to process exactly one argument (`arg1^`) and write the result of this argument into `dest^`.

```
procedure MySquareRoot(AnOperation: POperation); far;
begin
  with AnOperation^ do
    dest^ := sqrt(arg1^);
end;
```

and

```
Parser1.AddFunctionOneParam('squareroot', @MySquareRoot);
```

You can add functions dynamically at runtime and discard those too. For the latter operation use either `ClearFunctions` or `ClearFunction`.

AddFunctionTwoParam method

Applies To
TCustomParser, TParser

Declaration
```
procedure AddFunctionTwoParam( const AFunctionName: string;
                               const Func: TMathProcedure);
```

Description
This procedure will add under the name of `AFunctionName` the code pointed to by `Func`, where the function is declared as "`far`", see `TMathProcedure`.

The passed function code is expected to process exactly two arguments (`arg1^` and `arg2^`) and write the result of the arguments into `dest^`.

```
procedure MyMaximum(AnOperation: POperation); far;
begin
  with AnOperation^ do
    if arg1^ < arg2^ then
      dest^ := arg2^
    else
      dest^ := arg1^;
end;
```

and

```
Parser1.AddFunctionTwoParam('maximumvalue', @MyMaximum);
```

You can add functions dynamically at runtime and discard those too. For the latter operation use either `ClearFunctions` or `ClearFunction`.

ClearVariables method

Applies To
TCustomParser, TParser

Declaration
```
procedure ClearVariables;
```

Description
All variables are discarded from the parser. The internal representation of the expression is invalidated. You need to set the `Expression` property or call `ParseExpression` to continue using the parser.

ClearVariable method

Applies To
TCustomParser, TParser

Declaration
```
procedure ClearVariable(const AVarName: string);
```

Description
The variable as passed is discarded from the parser. The internal representation of the expression is invalidated. You need to set the `Expression` property or call `ParseExpression` to continue using the parser.

VariableExists method

Applies To
TCustomParser, TParser

Declaration
```
function VariableExists(const AVarName: string): boolean;
```

Description
This function tests whether a variable with the passed name exists. It returns True if a variable of the name exists, False if not.

ClearFunctions method

Applies To
TCustomParser, TParser

Declaration
```
procedure ClearFunctions;
```

Description
All functions are discarded from the parser. The internal representation of the expression is invalidated. You need to set the `Expression` property or call `ParseExpression` to continue using the parser.

ClearFunction method

Applies To
TCustomParser, TParser

Declaration
```
procedure ClearFunction(const AFunctionName: string);
```

Description
The function as passed is discarded from the parser. The internal representation of the expression is invalidated. You need to set the `Expression` property or call `ParseExpression` to continue using the parser.

OnParserError event

Applies To
TCustomParser, TParser

Declaration
```
property OnParserError: TParserExceptionEvent;
```

Description
In case the parsing engine hit an error an exception will be raised (see the declared exceptions). If `OnParserError` has been assigned the exception will be passed to the event handler and no further action will be taken.

If no event handler has been assigned the exception will fall through to the next exception handler and, if it is the application's default exception handler, be displayed in a message box.

TParser Component

Properties Methods Events

Unit
Parser10

Description
The only purpose of `TParser` is to publish the variable properties declared as protected in `TCustomParser` and to add default functions to the parsing engine.

Properties
Run-time only
Key properties

A	PascalNumberformat	
B	T	
C	Value	
D	Variable	
E	X	
Expression	Y	
LinkedOperationList	ParserError	

Methods
Key methods

AddFunctionOneParam	ClearVariable	ParseExpression
AddFunctionTwoParam	ClearVariables	RemoveBlanks
ClearFunction	FreeExpression	SetVariable
ClearFunctions	GetVariable	VariableExists

Events
Key events
OnParserError

RemoveBlanks method

Applies To
TParser

Declaration
```
class function RemoveBlanks(const s: string): string;
```

Description
This function will return the passed string with all blanks removed.

ParserFloat type

Unit
Parser10

Declaration
```
ParserFloat = double;
```

Description
`ParserFloat` is the generic floating point type of the component.

It is used in declarations of the properties and for direct memory access of variable values as well as for internal calculations.

Note: never use the old Borland / Turbo Pascal "real" type, only use the true floating point types single, double or extended.

PParserFloat type

Unit
Parser10

Declaration
```
PParserFloat = ^ParserFloat;
```

Description
`PParserFloat` is a pointer to a memory location containing a `ParserFloat`.

This pointer may be returned by `SetVariable` for very fast, direct variable access in memory.

TToken type

Unit
Parser10

Declaration
```
TToken = ( variab, constant,
           minus,
           sum, diff, prod, divis, modulo, IntDiv,
           integerpower, realpower,
           square, third, fourth,
           FuncOneVar, FuncTwoVar);
```

Description
`TToken` declares all possible tokens the parser understands.

Usually you will not need to use this type.

POperation type

Unit
Parser10

Declaration
```
POperation = ^TOperation;
```

Description
`POperation` is a pointer to a record of `TOperation`.

Usually there will be no need to ever use this type.

TMathProcedure type

Unit
Parser10

Declaration
```
TMathProcedure = procedure(AnOperation: POperation);
```

Description
Functions that are added to the engine must adhere to this declaration, see also the help topic for `POperation`; you also need to make sure that the procedure is declared "`far`" in Delphi 1. See `AddFunctionOneParam` and `AddFunctionTwoParam` for adding additional functions to the parsing engine.

If the function you are adding has been declared in the interface section of a unit, it implicitly has been declared as "`far`" by the compiler already (see the Delphi 1 compiler documentation).

There is no need to declare these functions "far" in 32bit (Delphi 2 / 3, C++ Builder), as all functions are automatically declared far in these systems.

TOperation type

Unit
Parser10

Declaration
```
TOperation = record
               Arg1, Arg2: PParserFloat;
               Dest: PParserFloat;
               NextOperation: POperation;
               Operation: TMathProcedure;
               Token: TToken;
             end;
```

Description
`TOperation` is used when parsing an expression. It is used internally only.

TParserExceptionEvent type

Unit
Parser10

Declaration
```
TParserExceptionEvent = procedure ( Sender: TObject;
                                    E: Exception) of object;
```

Description
If an error occurs during parsing, the `OnParserError` will be fired. The method called will be declared according to the template defined by `TParserExceptionEvent`.

We could use the Forms unit here and the `TExceptionEvent` declared therein, but that would give us all the VCL overhead. To avoid this we consequentially just re-declare an appropriate event.