Parsing and Evaluating Mathematical Expressions in Object Pascal

Michel Deslierres
michel.deslierres@sigmdel.ca
2016-07-28
http://www.sigmdel.ca

Version 1.0

There are two versions of this text on parsing and evaluating mathematical expressions. The original and incomplete version describes parsers written with Embarcadero's Delphi. It is still available. This version describes the source code that compiles with Free Pascal 2.6.4 or better.

The two versions are not much different. Porting to Free Pascal required treating floating point exceptions at the very beginning. National language support is also different. Finally, dynamic arrays are used in the Free Pascal version which would mean that the code would not compile under the now very old version 2 (circa 1995) of Delphi.

Table des matières

1 Introduction....3
1.1 Translating Text....3
1.2 Source Code and License....5
2 Parsing a Simplified Grammar....7
2.1 Simple Grammar....7
2.2 Recursive Descent Parsing....9
2.3 Implementation....9
2.4 Tokens....13
2.5 Errors....16
2.6 Floating Point Exceptions....19
2.7 User Locale....22
2.8 Conditional directives....23
2.9 Demonstration Program....24
2.10 Testing....25
3 More Mathematical Operations and Comments....27
3.1 Changes to the tokenizer....27
3.2 Wirth Syntax Notation....30
3.3 Integer Division....31

3.4 Exponentiation....33
3.5 Precedence....34
3.6 Universal Conversion of Expressions....35
4 Adding Constants, Functions and Variables....37
4.1 Identifiers....37
4.2 Built-in Functions....39
4.3 Parsing a Built-in Function....44
4.4 List Separator....47
4.5 Missing Mathematical Functions....48
4.6 Variables....48
4.7 Target Hint....52
5 Extended National Language Support....55
5.1 Localization in Free Pascal....55
5.2 Universal Translations of Expressions....56
5.3 Duplicate names....59
5.4 Caleçon....61
6 Appendix: List of built in functions, constants and keywords....63
7 Bibliography....65

1 Introduction

Modern general purpose operating systems all seem to include an application which imitates more or less a hand held calculator that can be invoked by a user that needs to calculate a value. In some circumstances this is not sufficient, and a programmer needs an expression evaluator in a program that allow users to specify a mathematical expression. The Object Pascal code presented here parses and evaluates such expressions given as strings such as '`3+4*(27-9/2)`' and '`e^sin(3*pi/4)`'and returns their numerical value (93 and 2.02811498164747 respectively).

1.1 Translating Text

Breaking up text into elements or tokens and distinguishing numbers (such as `27` and `2`), identifiers (such as `pi` and `sin`) and operators (such as `+` and `*`) is called lexical analysis. In the present context, this is a simple task because the type of token can be inferred from its first or, occasionally, first two characters.

The more difficult problem is the syntactical analysis or parsing of the lexical elements. In this step, numbers, identifiers and operators are arranged according to the rules defining mathematical operations so as to be easily evaluated. The last task is the actual calculation.

While the process can be logically divided into three steps, in practice it need not be. Evaluation of an expression can be done all at once. Here, lexical analysis will be performed separately from the rest. Parsing and evaluating can easily be combined and that is the first approach that will be considered in the following papers. When calculations must be done repeatedly as when drawing the curve defined by a function, say `x*sin(x)`, it is best to perform the lexical and syntactic analysis of the expression only once and evaluate many times as the value of `x` is modified.

There is more than one way to implement this two step strategy. The avenue chosen here uses a tree representing all the elements of the formula (numeric constants, calls to predefined functions or constants such as `sin` and `pi`, etc.) and the operations that are to be performed in their correct order. One of the advantages of this approach is that it makes it relatively easy for the programmer to add more internal functions, constants and variables. It is also relatively easy to include user defined functions, that is external functions that are defined by a user of a program at run time.

The material will be presented in many parts. The first four parts are concerned with creating a one pass parser suitable for evaluating user specified mathematical expression one at a time.

1. Parsing a Simplified Grammar

In the first paper, a simple grammar that contain numbers, parenthesis, and the four arithmetic operators + − * and / with the usual precedence rules is presented. This grammar contains only two type of lexical elements: numbers and operators. A simple tokenizer capable of recognizing these types of elements is constructed. Then a one pass parser and evaluator is added.

2. More Mathematical Operations and Comments

   The grammar is expanded to include exponentiation, integer division and modulo operations and C-style end of line comments. This will require a small change to the tokenizer to recognize new operators and of course changes to the parser. The expanded grammar will be defined with a more rigorous notation. Optional [] and {} parenthesis are also added.

3. Adding Constants and Functions

   Built in constants, pi = 3,141... and e = 2,718..., and functions such as sin(), abs() are added. The mechanism chosen is straightforward and programmers can easily add more. At the same time the keyword mod and div are added as synonyms for the % and : operators introduced in the previous section. There is the addition of a second list of call targets which makes it easier to implement user defined constants and variables. The grammar is augmented to add statements that creates or removes such variables.

4. Extended National Language Support

   National language support is added to the parser. This article can, for the most part, be skipped. Its content is not directly concerned with the subject of parsing mathematical expressions and no substantial change is made to the parser. A package file is provided for easy integration with the Lazarus IDE. The result is the parser used in the application **Caleçon**.

An upcoming second series of chapters will look at constructing a two pass evaluator of mathematical expressions. The first pass constructs a parse tree, the second pass evaluates the tree.

5. Constructing a Parse Tree

   A new parser is introduced. It constructs a parse tree in memory which is a representation of syntactic elements as nodes. The tree can easily and rapidly be evaluated. To make this useful, programmer defined variables are added. There is no change to the grammar and the same tokenizer is used. And it turns out that it is simple to add user defined constants and functions.

6. Adding Boolean Operations

Adding the `if()` function, Boolean comparisons (=, >, >= and so on) and Boolean operations (or, and, xor). Boolean comparisons are a new syntactical element that needs to be added to the grammar. The boolean operations require minimal changes to previously defined syntactic elements of the gramme. The tokenizer must also be changed.

7. Optimizing by Constant Folding

   This paper discusses two extensions. The simpler is cloning a parse tree, which is a “deep” copy of the original tree so that it and it’s clone do not share any nodes. More complex, is the optimization of a parse tree so that an expression such as `sin(3*x*pi/2)` will be simplified to replace the multiplication by 3 and division by 2 with the multiplication by 1.5.

1.2 Source Code and License

All the source code for the various versions of the parsers and for the accompanying demonstration programs developed by the author is released under the terms of BSD style license:

For programmers that might want to use the code, this license

> ... is effectively a statement that you can do anything with the program or its source, but you do not have any warranty and none of the authors has any liability (basically, you cannot sue anybody). This new BSD license is intended to encourage product commercialization. Any BSD code can be sold or included in proprietary products without any restrictions on the availability of your code or your future behaviour.
>
> Do not confuse the new BSD license with “public domain”. While an item in the public domain is also free for all to use, it has no owner.
>
> Bruce Montague (2013)

Accordingly, use this code if you find it useful, just don’t pass it off as your work. Furthermore, suggestions or constructive critiques are most welcome. All the source code for the various versions of the parsers is available free of charge. According to the license you could sell it to someone, but why someone would pay for something that is available for free is hard to understand and it is rather unethical that someone would want to profit in such a way.

2 Parsing a Simplified Grammar

There is no need to work out everything anew. Much is available on building parsers even when looking only at material using Pascal. Completely worked out parsers such as **JEDI Code Library** (JCL)'s unit JclExprEval.pas, Renate Schaaf's TExpress Version 2.5, TParser version 10.1 by Renate Schaaf, Alin Flaider and Stefan Hoffmeister or symbolic.pas by Marco van de Voort are easily found.

Because it is difficult to follow the logic in such complex units, the approach used here, starts with a much simpler parser. It will be able to handle expressions that contain numbers, parenthesis ( ) and the four arithmetic operators + − * and / with the usual precedence rules:

a) anything included in parentheses is evaluated first,
b) multiplications and divisions are evaluated next and finally
c) addition and subtraction are done last.

Complications such as exponentiation, internal functions (sin, cos etc.), predefined constants (e and pi for example) and variables (*x*, *t* for example) and user defined functions and variables will be added later.

2.1 Simple Grammar

An older book, Kernighan and Plauger's *Software Tools in Pascal*, contains code for a macro processor with an expression evaluator for the simple grammar loosely defined above. This is approximately how the authors define the syntax of mathematical expressions (p. 298):

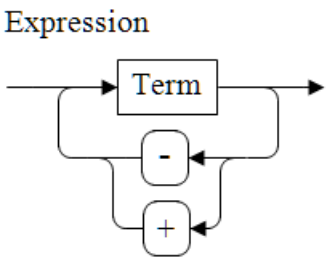
expression : term | term + term | term − term

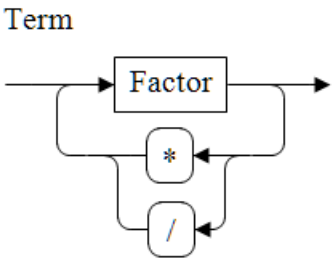
term : factor | factor * factor | factor / factor

factor : number | (expression) | + factor | − factor

the vertical bar | denotes 'or' while the operators + − * and / are literals.

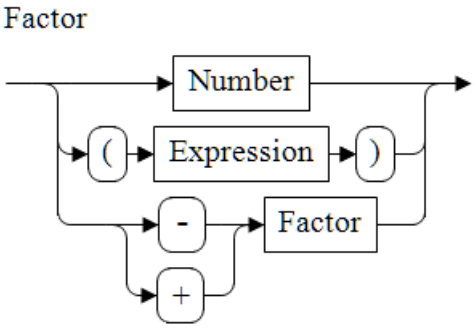
This grammar in pseudo Backus-Naur Form can be specified using syntax diagrams, also called railroad diagrams, as found in Jensen and Wirth's *Report on Pascal* (1975).



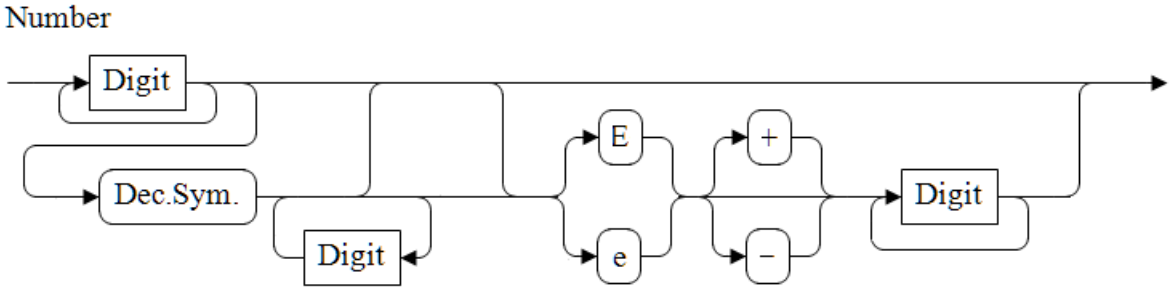


It is now very clear that an expression is a term or a sequence of terms separated by the + or − operators. Similarly a term is a factor or a sequence of factors separated by the * or / operators. A factor is a number which optionally can be preceded by a leading (unary) + or −.

A factor can also be an expression enclosed in parenthesis. The unary + operator is redundant and thus ignored. The unary – operator is equivalent to multiplication by −1. Thus the factor '−32' is equal to the term −1 * 32.



It remains to define what a number is. It can be an integer or a real number represented in decimal notation or floating value notation.



While this may look complex, the diagram corresponds to the usual definition of a number used in programming languages, spreadsheets or even calculators. Digit is any character 0, 1 to 9. Dec.Sym. is the locale dependant decimal symbol; usually the period or the comma. Here are examples of well-formed numbers where it is assumed that the decimal symbol is the period '.':

1.83	1.83E0	0.183E1	0.0183E2
183	1.83E2	18.3E1	183E0
0.183	1.83E-1	18.3E-2	183E-3

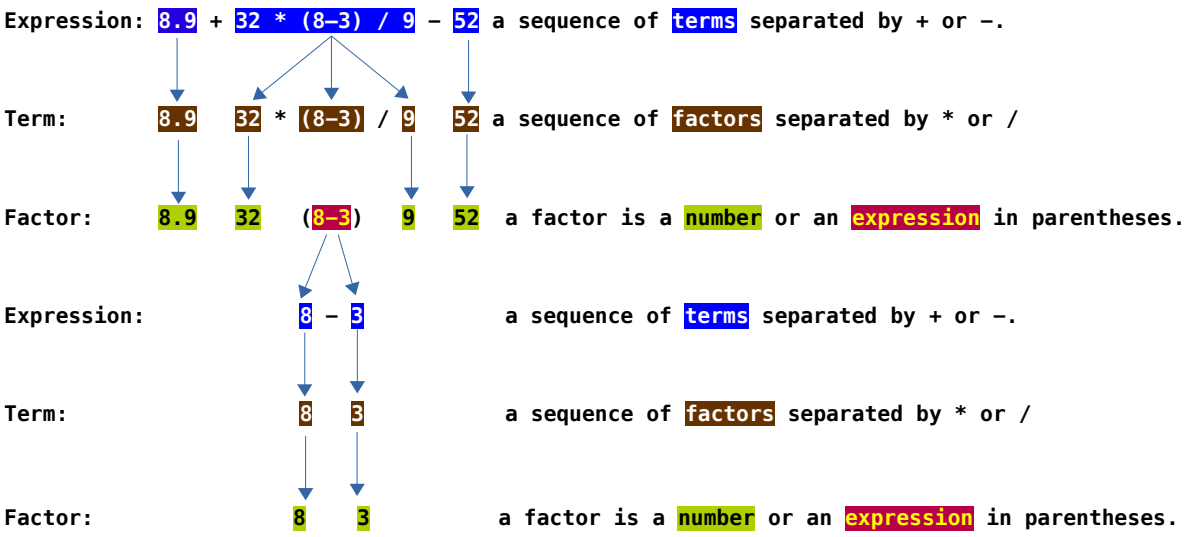
On the other hand, 1.83E*8 will be deemed a badly formed number; at least one digit must follow the exponent identifier E and the sequence will be scanned as the invalid number 1.83E the operator * and the number 8.

The definition of a number is such that it can be any real number including any whole

numbers. However, all numbers, either whole or not, will be of type `float` which is declared in the unit ParserTypes.pas. Currently, `float` is declared to be an `extended` real number. This number type is 10 bytes wide, has a range from 3.6×10^{-4951} to 1.1×10^{4932} with about 20 significant digits. In some circumstances it may be better to change `float` to be a `double`. Doubles are 8 bytes wide and have the range 5.0×10^{-324} to 1.7×10^{308} with about 15 significant digits. The format for double is almost universal. See Goldberg (1991) for more information about the IEEE 754 Standard.

2.2 Recursive Descent Parsing

The above language can be handled quite naturally by what is called a recursive descent parser. The expression `8.9 + 32*(8 - 3) / 9 + 52` is broken down into its constituent parts using that type of parser:



Note how eventually everything has to resolve into a number. How else could a numeric expression be calculated ? Evaluation is done in reverse order once all numbers have been found. Hence the expression in parentheses is calculated first. Then multiplications and divisions are performed and additions and subtractions are done last. As can be seen in the example, parsing an expression can involve parsing an expression contained within. The same is true from term and factor. This is the reason why parsing is said to be recursive.

2.3 Implementation

In Kernighan and Plauger's parser each syntactic element (expression, term, and factor) has a corresponding function which returns an integer value. Here is the expression function:

```
function Expression(var src: string; var i: integer): integer;
var
  v: integer;
  t: char;
begin
  v := Term(src, i);
  t := NextChar(src, i);
  while (t in ['+', '-']) do begin
    i := i + 1;
    if t = '+' then
      v := t + Term(src, i)
    else if t = '-' then
      v := t - Term(src, i);
    t := NextChar(src, i);
  end;
  Expression := v;
end;
```

Since expressions are sums or differences of terms, the function begins by looking for a first value using the term function. Assuming a first term has been found, it then looks for a '+' or '-' operator. If one is found then a second term is added or subtracted from the first term. The process continues as long as successive terms are separated by a '+' or '-'.

Kernighan and Plauger mix parsing and scanning. The function `NextChar(const src: string; var i: integer)` returns the character in the string `src` at index `i`. If the character at position `i` is a space, tabs, and other meaningless character (often called white space), then `i` is first incremented until a non-white space character is found in `src`. The Expression function advances the index `i` over the '+' or '-' operators it finds. To make expansion easier, the two will be separated. Here is a version of our simple parser's `Expression` function presented in a fashion as close to Kernighan and Plauger's as possible. Note that in our version `Expression`, `Term` and `Factor` return real values of type `float`.

```
function Expression: float;
begin
  result := Term;
  while TokenType in [ttPlus, ttMinus] then begin
     if TokenType = ttPlus then begin
       NextTrueToken;
       result := result + Term;
     end
     else begin // TokenType = ttMinus
       NextTrueToken;
       result := result - Term;
     end
  end;
end;
```

In keeping with current usage the internal variable `result` is used instead of the explicit `v` variable. The tokenizer keeps track of its position within the source string, so the parameters `src` and `i` do not have to be passed between the functions `Expression`, `Term` and `Factor`.

The tokenizer also makes available the type of the current token so there is no need to save it in a local variable `t`. It's up to the parser to tell the tokenizer to move on to the next token. The tokenizer's method `NextTrueToken` moves to the next token skipping white space. The call to move on to the next token has to be done after the second check of the current token's type and before the second call to `Term`.

It's a very simple matter to adapt `Expression` to get the function `Term`. The implementation below is the actual function which uses an endless repeat loop instead of a while loop to avoid an occasional extra logical test.

```
function Term: float;
begin
  result := Factor;
  repeat
    if TokenType = ttMult then begin
      NextTrueToken;
      result := result * Factor;
    end
    else if TokenType = ttDiv then begin
      NextTrueToken;
      result := result / Factor;
    end
    else
      exit;
  until false;
end;
```

There remains only `Factor`. If it encounters a number, it's string representation, held in the tokenizer's `Token` property will be evaluated to a floating point value by a function called `sfloat`. If the string is not a well-formed number, that function will signal an error. If `Factor` encounters a unary '+' as in the formula `+(3-4)*8`, this redundant + is simply ignored, and `Factor` moves on to the next token and next factor. If `Factor` encounters an unary '-' as in the formula `-(3-4)*8` it looks for a factor starting at the next token and then returns the negative of the value of that factor.

```
function Factor: float;
begin
  result := 0;
  if TokenType = ttNumber then begin
    result := sfloat(Token);
    NextTrueToken;
  end
  else if TokenType = ttPlus then begin
    Tokenizer.NextTrueToken;  // skip and ignore leading '+'
    result := Factor();       // dont forget () in fpc
  end
  else if TokenType = ttMinus then begin
    NextTrueToken;  // skip '-'
```

```
    result := - Factor();      // unary -
  end
  else if TokenType = ttLeftParenthesis then begin
    NextTrueToken;  // skip '('
    result := Expression;
    if TokenType <> ttRightParenthesis then begin
      ParserError(peExpectedRightPar);
    end;
    NextTrueToken; // skip ')'
  end
  else if TokenType = ttEOS then
    ParserError(peUnexpectedEOS)
  else
    ParserError(peUnexpectedElement);
end;
```

Note: The procedure calls on itself. It is necessary to add the empty parenthesis (empty parameter list) to `Factor` in a **Free Pascal** unit that is in `objfpc` mode. In **Delphi** or in **Free Pascal** in `delphi` mode, `result := Factor;` within `Factor` will invoke the procedure recursively, but in `objfpc` mode `result := Factor;` is equivalent to `result := result;` (or `Factor = Factor;` for that matter) which hardly does what is needed. This is documented in *Free Pascal/Lazarus gotchas* (Deslierres 2016) along with other similar differences between the two dialects of Object Pascal.

If `Factor` encounters a '(' then it will return the value of an expression starting with the next token. Once the expression has been parsed, `Factor` checks for the matching right parenthesis signalling an error if it is not found. If the end of the source has been found, then an error is signalled. Finally, if anything else is found, then `Factor` will signal an error as it does not know how to handle it.

Each of the three function performing syntactic analysis begins by checking the current token's type which means that the parser has to be 'primed'. To evaluate a formula, the tokenizer's `NextTrueToken` method must first be invoked to get the first token's type. Then the result will be obtained by calling `Expression`:

```
function EvaluateExpression(const aSource: string): float;
begin
  Source := aSource;
  NextTrueToken; // prime the pump
  result := Expression;
  if TokenType <> ttEOS then
    ParserError(peExpectedEOS);
end;
```

Thus the parser always starts with the sequence

`Expression` → `Term` ⟶ `Factor`.

That sequence explains why multiplication is done before addition; the function `Term` (which handles the * and / operators) is executed before the function `Expression`. (which handles the + and – operators). Since Factor is called last, it is executed first which means that

expressions in parentheses are calculated first. Operator precedence is handled automatically.

Once an expression has been parsed and evaluated, a check is made to ensure that there is nothing left in the source.

Those four functions are the heart of the whole parser. The details of scanning the mathematical expression will now be examined.

2.4 Tokens

The string containing the mathematical expression to be evaluated is broken up into tokens that are the smallest units of meaning understood by the parser. Tokens are to a mathematical expression what words are to a sentence.

A tokenizer could perform its task all at once, returning an array of tokens. Instead, the tokenizer can return one token at a time. The parser requests the next token once it has dealt with the current one. It is up to the tokenizer to keep track of its position within the source code. While not mandatory, an object with its capacity to have fields and its ability to hide some of them, is well suited for handling these tasks.

The tokenizer identifies the type of the tokens it finds. For that reason, some would call the tokenizer a lexer. The current grammar is so simple that a token's type is recognized by looking at its first character. Furthermore, because the recognized types of tokens have very specific constraint about the characters they may contain, the tokenizer identifies adjacent tokens even if they are not separated by spaces. Hence the string '2*41' will be broken into three tokens : '2', '*', and '41' because numbers which start with a digit do not contain the character '*' which is an operator. Fundamentally there are three kinds of tokens in this simple context:

1. A *white space sequence* must begin with a space or a predecessor, a character with a code lower than the space character . The tokenizer will accumulate all contiguous characters with a code less than or equal to 32 and greater than 0 in the current token.
2. A *number* must begin with a digit . A number can be an integer or a real number. The tokenizer will attempt to accumulate in the current token the biggest number possible so that the string '1234' will be interpreted as a single number 1234 and not two numbers 1 and 234 or 12 and 34 or even four numbers 1, 2, 3 and 4.
3. An *operator*, which is a one character token as given in the following list. The one character tokens are "`-+*/()`" excluding the quotation marks.

Anything else will be a one character token of unknown type and hence illegal while each operator, also of length 1, is deemed a token type.

```
TMathTokenType = (
```

```
    ttUnknown,            //  unknown (illegal) token
    ttWhite,              //  white space; ascii code 1 to 32
    ttNumber,             //  integer or real number
    ttEOS,                //  end of source
    ttMinus,              //  '-'
    ttPlus,               //  '+'
    ttMult,               //  '*'
    ttDiv,                //  '/'
    ttLeftParenthesis,    //  '('
    ttRightParenthesis); //  ')'
```

The tokenizer has additional fields or properties in Object Pascal parlance. The most important are

```
property Source: string;           // the formula to parse
property Token: string;            // the current token
property TokenType: TMdTokenType; // the type of the current token
```

Other properties are present because they help in reporting errors to the user

```
property ErrorCode: TMathError; // meNoError if all is well
property ErrorMessage: string;  // empty if ErrorCode = meNoError
property TokenStart: integer;   // Index of the current token
property TokenLength: integer;  // Length of the current token.
```

Except for `Source`, all these properties are read only and are updated by calls to the functions

```
function NextTokenType: TMdTokenType;
function NextTrueTokenType: TMdTokenType;
```

which each returns a token type. Translating will use the first thus preserving useful but the syntactically meaningless white space. Parsing uses the second of these functions which skips meaningless white space thus never returning a `ttWhite`.

There are two functions that assist in conversions of numbers from string to float values and back in accordance with a local format settings property.

```
function sfloat(const value: string): float;
function floats(const value: float): string;
property FormatSettings: TFormatSettings;
property DecimalSeparator: char;
```

The conversion functions use the decimal symbol of the format settings. The property `DecimalSeparator` gives direct access to that field.

The tokenizer uses three integer variables to define tokens: `FTokenStart` which is the position in the source string of the first character of the current token, `FTokenLength` which is the length of the current token and `FNextCharPos` which is the start of the next token after the current one. That field is redundant since `FNextCharPos = FTokenStart + FTokenLength`, but it will prove useful in the future when two character operators are

introduced.

Here is the situation after the formula is stored in the `Source` field of the tokenizer:

```
FSource = 8.9 + 32*(8 - 3) / 9 + 52
          ↑
          FNextCharPos = 1
          FTokenStart  = 1
          FTokenLength = 0
```

The function `NextTokenType` returns `ttNumber` which it identified with digit '8' at position `FTokenStart` set equal to the initial `FNextCharPos`. The latter is then incremented as the number is scanned. Once the space is reached, the number token has been identified:

```
FSource = 8.9 + 32*(8 - 3) / 9 + 52
          ↑  ↑
          ↑  FNextCharPos = 4
          FTokenStart     = 1
          FTokenLength    = 3
```

The next call to `NextTokenType` identifies a white space token of type `ttWhite` of length 1:

```
FSource = 8.9 + 32*(8 - 3) / 9 + 52
             ↑↑
             ↑  FNextCharPos = 5
             FTokenStart     = 4
             FTokenLength    = 1
```

The third call to `NextTokenType` identifies a white space token of type `ttPlus` of length 1:

```
FSource = 8.9 + 32*(8 - 3) / 9 + 52
              ↑↑
              ↑  FNextCharPos = 6
              FTokenStart     = 5
              FTokenLength    = 1
```

The code for NextTokenType is pretty straightforward:

```
var
  c: char;
begin
  // move past current token
  FTokenStart := FNextCharPos;
  FTokenLength := 0;
  if FNextCharPos > length(FSource) then
    result := ttEOS
  else begin
    c := FSource[FTokenStart];
    if isWhite(c) then
      GetWhite
```

```
    else if isDigit(c) then
      GetNumber
    else
      GetOperator(c);
  end;
  FTokenType := result;
end;
```

The role of the binary functions `isWhite` and `isDigit` is obvious. The same can be said by the procedures `GetWhite`, `GetNumber` and `GetOperator`.

2.5 Errors

The above blissfully ignored errors except for invoking an as yet undefined `ParserError` routine. Unfortunately, error handling is at least as complicated as the main business at hand. There are two categories of errors that can occur when evaluating a mathematical expression. Errors are encountered when parsing an invalid expression. It is impossible to make sense of a statement such as '2+*8' with an invalid syntax. On the other hand, '2/(1-1)' can be parsed because it is syntactically correct but it cannot be evaluated because it involves division by 0.

Syntax errors will be found in the private functions `Expression`, `Term` and `Factor` and in the public function `EvaluateExpression`. Non-syntactic errors can be encountered when performing calculations in the private functions: an overflow error could occur in `Expression` or `Term` while a division by 0 can arise in `Term` only. While unlikely, there is a possibility of non-mathematical run-time errors such as running out of heap memory. These possible run-time errors are more difficult to handle than syntactic parsing errors. The simplest way of handling this is to use **Free Pascal**'s exception handling. Since exceptions are used to signal run-time errors, it seems easier to signal a syntax error encountered while parsing by raising an exception.

```
type
  { Parser exception }
  EParserError = class(Exception)
  public
    Error: TMathError;
    TokenStart: integer;
    TokenLength: integer;
  end;
```

The exception's `Message` field will contain a short text identifying the problem. In addition, the exception will specify the error code and its location which only makes sense when a parser syntax error has occurred. Error codes are member of an enumerated type `TMathError` also defined in ParserTypes.pas .

There results something of a mess with this approach as many types of exceptions can be raised when evaluating a mathematical expression. Use of this parser would be simpler if

`EParserError` were the only type of exception raised. This is accomplished by recasting other types of exceptions into `EparserError`'s while preserving the original error message.

Before looking at the details, there is another problem with error handling that must be tackled. How should errors be reported to the user? The answer depends on what a program using the parser is trying to achieve.

Consider the demonstration program supplied with this parser. The user enters a mathematical expression and expects a numeric value as an answer. In that case, immediate feedback is necessary no matter the source of the exception. This can be done in one of two ways. The simpler is to let the exception handling mechanism take care of everything. That means a window will pop up informing the user of the exception and giving him or her a choice of what to do. Another possibility is to display a message in the calculator informing the user as to the problem. In that case, it would be necessary to handle the exception that could be raised in `EvaluateExpression`. Instead of putting the burden on code using the parser, `EvaluateExpression` will handle exceptions by default.

All this is accomplished by adding two members to the `TMathError` enumeration to cover `EMathError` class exceptions and any other run-time exceptions.

```
type
  { Errors that the mathematical expression evaluator may report.}
  TMathError = (
    meNoError,            // never returned by an exception
    meUnexpectedEOS,      // unexpected end of source such as "5 +"
    meExpectedEOS,        // extra bits found as in "5 + 8 32"
    meInvalidNumber,      // as is "5.2E/8"
    meUnexpectedElement, // as in 5+*8 - the * is unexpected
    meExpectedRightPar,   // missing right matching parenthesis
    meEvalError,          // an exception of type EMathError
    meOtherError);        // other run-time exception

  TParserError = meUnexpectedEOS..meExpectedRightPar;

const
  ParserErrors: set of TMathError =
[low(TParserError)..high(TParserError)];
```

Syntactic errors are now a sub-range of type `TParserError`. The tokenizer contains a flag

```
property RaiseExceptions: boolean read FRaiseExceptions write
FRaiseExceptions;
```

that indicates if the parser ultimately pases exceptions to the application or if errors are signalled in the `ErrorCode` property which the application must then poll and handle. Two protected procedures are added to the tokenizer to signal errors: `ParserError` which has already been encountered and the following method that `ParserError` uses:

```
procedure TMathTokenizer.Error(Code: TMathError; const msg:
string);
var
  E: EParserError;
begin
  // store local copies of the error code and message
  FErrorCode := code;
  FErrorMessage := msg;
  if RaiseExceptions or (Code in ParserErrors) then begin
    // raise the exception
    E := EParserError.Create(msg);
    E.Error := Code;
    E.TokenStart := TokenStart;
    E.TokenLength := TokenLength;
    Raise E;
  end;
end;
```

It may seem strange that when a parser error is signalled an exception is raised even if the `RaiseException` flag is false. Nevertheless, it has to be so otherwise parsing would continue after a syntax error has been found and eventually some bogus value would be returned. Parser exceptions along with run-time exceptions are handled in `EvaluateExpression` which is somewhat more complex than what was shown previously:

```
function TMathEvaluator.EvaluateExpression(const aSource: string):
float;
begin
  Source := aSource; // resets error state
  NextTrueToken;     // prime the pump
  try
    result := Expression;
    if (TokenType <> ttEOS) then
      ParserError(meExpectedEOS);
  except
    On E: EParserError do begin
      if RaiseExceptions then
        Raise;
    end;
    On (E: Exception do begin
      SelectAll; // select all the source
      if (E is EMathError) or (E is EDivByZero) then
        Error(meEvalError, Format(SmeEvalError, [E.Message]))
      else
        Error(meOtherError, Format(SmeOtherError, [E.Message]));
    end;
  end;
end;
```

Exceptions of type `EParserError` are caught in the try-except pair and passed on to the caller only if `RaiseExceptions` is true. All other exceptions are also caught. The first step

in handling them is to call on the tokenizer's `SelectAll` method which sets the whole source string as the current token. The `Error` method, already presented, is invoked with the correct error code and error message. It will in turn raise `EParserExceptions` if `RaiseExceptions` is true.

There are two division by 0 exceptions defined in SysUtils.pas.

```
    { integer math exceptions }
    EInterror     = Class(EExternal);
    EDivByZero    = Class(EIntError);
...
    { General math errors }
    EMathError  = Class(EExternal);
    EZeroDivide = Class(EMathError);
```

Since we are not performing integer arithmetic, it would seem logical that division by zero would raise an `EZeroDivide` exception of type `EMathError`. Currently (FPC version 3.0 / Lazarus version 1.7) floating point division by 0.0 raises an `EDivByZero` exception which is of type `EIntError`. This is the reason for the `if (E is EMathError) or (E is EDivByZero)` test.

To summarise:

- the tokenizer's `NextToken` and `NextTrueToken` methods does not generate exceptions, they signal errors by returning a `ttUnknown` token type;
- the parser's `Term`, `Expression` and `Factor` methods generate exceptions when errors are encountered;
- the parser's `EvaluateExpression` handles all exceptions and passes them on to the application (as `EParserError`s) or signal errors in the `ErrorCode` field according to the `RaiseExceptions` property.

It would be nice if that were the end of the story but unfortunately there remains one problem to consider.

2.6 Floating Point Exceptions

The error handling is not yet complete. Envision a program that plots graphs of functions defined by the user. This is done by calculating the value of the function at many points in its range. It would not be practical to report every overflow or invalid operation exception to the user as proposed above. The program itself should handle these points. Consequently, the programmer using the parser should decide which mathematical exceptions will be raised.

This is done by setting the floating point exception mask; something easily done in the math.pas unit. Including `exZeroDivide` in the mask means that '5/0' will return the non-

number `+Inf` and will not raise an exception. Similarly, including `exInvalidOp` in the mask means that '0/0' will return `NaN` (not a number) and execution will continue as if a valid operation had been performed.

Unfortunately, mucking about with the FPU exception mask, (also called the 8087 control word in earlier times when the FPU was a separate chip) can result in all sorts of problems. In fact, **Lazarus** does not set the same initial floating point exception mask on all platforms. This is discussed on the **Lazarus forum** (2011), in the **Lazarus bugtracer** (2011) and in the **Lazarus wiki** (2016). **Free Pascal** run time library initializes the mask to `[exDenormalize, exUnderflow, exPrecision]`. But the mask is changed in the **Lazarus Component Library** in in the following routine invoked early on in the `TApplication.Initialize` procedure:

```
procedure TGtk2WidgetSet.AppInit(var ScreenInfo: TScreenInfo);
begin
  {$if defined(cpui386) or defined(cpux86_64)}
  // needed otherwise some gtk theme engines crash with division
by zero
  {$IFNDEF DisableGtkDivZeroFix}
    SetExceptionMask(GetExceptionMask +
[exOverflow,exZeroDivide,exInvalidOp]);
  {$ENDIF}
  {$ifend}
  ...
```

The solution used is as proposed in the wiki: the exception mask is set to a desired value just before evaluating an expression, and then it is reset to its previous value. Hence as far as the widget set is concerned, the fpu exception mask is not changed. This is actually a little more complex than it sounds because it is essential to restore the mask before any exception handling routine is called.

```
function TMathEvaluator.EvaluateExpression(const aSource: string):
float;
begin
  Source := aSource; // resets error state
  FOldExceptionMask := Math.SetExceptionMask(FExceptionMask);
  NextTrueToken;     // prime the pump
  try
    result := Expression;
    if (TokenType <> ttEOS) then
      // some extra bits found in the source, signal error
      ParserError(meExpectedEOS);
  except
    // pass on exception only if RaiseExceptions is true
    // converting all general (non parsing exceptions) exceptions
to
    // type EParserError. Restore the fpu exception mask before
    // passing on the exception.
```

```
    On E: EParserError do begin
      if RaiseExceptions then begin
        Math.SetExceptionMask(FOldExceptionMask);
        Raise;
      end;
    end;
    On E: Exception do begin
      SelectAll; // select all the source
      Math.SetExceptionMask(FOldExceptionMask);
      // distinguish math exceptions from other exceptions and let
      // the Error method re raise exception or not according to
the
      // RaiseException property.
      if (E is EMathError) or (E is EDivByZero) then
        Error(meEvalError, Format(SmeEvalError,
[lowercase(E.Message)]))
      else
        Error(meOtherError, Format(SmeOtherError,
[lowercase(E.Message)]));
    end;
  end;
  // normal exit after parsing and evaluating an expression
without error;
  // restore the privious exception mask.
  Math.SetExceptionMask(FOldExceptionMask);
end;
```

This is why the old exception mask is restored at three points in the routine. At the end, as expected, once evaluation has occured without problem. But also in the two exception handlers before execution is passed on to the application through a re-raised exception or not.

All this fiddling with the fpu exception mask is the default behaviour unless a `NO_EXCEPTION_MASK` directive is defined. In the past, I found that it was not necessary to go through this exercise for applications that I wrote for **Windows**. Presumably, if the parser is to be used in a **Windows** application, it would be possible to define `NO_EXCEPTION_MASK`. However, it would probably be safer to define the directive. There is enough discussion on the Web about changes made to the fpu mask by dlls or device drivers causing all sorts of hard to pinpoint difficulties to warrant being a little bit paranoid here.

Two final remarks about math exceptions. First, the default local fpu mask is set to `[exDenormalize, exUnderflow, exPrecision]` which is the same as the mask set by the **Free Pascal** run time library. Hence, the actual fpu mask used for calculations will be the same no matter which widget set is used[1]. Second, unmasking the precision exception is sure to crash a program using the Gtk2 widget set. Consequently, the exception mask setter protects against that possibility:

```
procedure TMathEvaluator.SetExceptionMask(value:
```

1 That is an unverified claim.

```
TFPUExceptionMask);
begin
  FExceptionMask := value {$IFDEF LCLGTK2} + [exPrecision]
{$ENDIF};
end;
```

2.7 User Locale

As indicated above, some decision had to be made about the handling of the decimal symbol. In English, the symbol is the period '.', in French it is the comma ','. Accordingly 5/2 written as a decimal number should be entered as 2.5 on an English language system and 2,5 on a French language system. This is precisely how **Excel** and **OpenOffice/LibreOffice.Calc** work. However, some mathematical applications such as **Maple** and **Euler Mathtool Box**, some statistical packages such as **Minitab** and (older versions of) **Gretl** and some graphing programs such as **Graph** and **Crispy Plotter** insist on using the period as the decimal symbol no matter what the locale.

The approach adopted here is to use the locale defined symbol by default but to allow change, even at run time. There is a `DecimalSeparator` field in the tokenizer which is set to the `DecimalSeparator` field found in SysUtils' `DefaultFormatSettings`. In fact the complete `DefaultFormatSettings` is copied to a `FormatSettings` property. The tokenizer's `sfloat` and `floats` methods use this local local format setting record.

There are platform differences when it comes to format settings. In **Windows**, `DefaultFormatSettings` is automatically set to the user's locale. In **Linux**, this is not the case; the unit clocale.pp must be included in the application's source file (*.lpr) as one of the first units used. If clocale is not included, then `DefaultFormatSettings` will be those of a US system no matter what the users's locale is. The situation in **Darwin** (**OS X** and **IOS**) is nominally the same as in **Linux**: `DefaultFormatSettings` is not updated to reflect the user's locale by the **Lazarus/Free Pacal** LCL and RTL. Two freepascal.org wiki pages (*OS X Programming Tips* and *FPC New Features 3.0*) suggest that clocale can pick up the user's locale from the unix-layer. I have not observed that behavior when running **FPC** 2.6.4. on **OSX 10.8.5 (Mountain Lion)**. However, as of version 2.7.1 the **Free Pascal** compiler has added a unit called iosxlocate that purports to get locales settings from the System Preferences. This has not been checked.

Here is the start of the demonstration application (demo1.lpr) showing how `DefaultFormatSettings` is updated to reflect the user's locale:

```
program demo1;

{$mode objfpc}{$H+}

uses
```

```
  {$IFDEF UNIX}
    {$IFDEF UseCThreads}cthreads,{$ENDIF}
    clocale,     // update DefaultFormatSettings
    {$IF defined(DARWIN) and (FPC_FULLVERSION>=2710)}
    iosxlocale, // use System Preferences
    {$ENDIF}
  {$ENDIF}
  Interfaces, // this includes the LCL widgetset
  ...
```

Unfortunately, there are more complications. Some format settings can cause problems. Consequently, correction is made to the local copy of the format settings in the tokenizer:

```
constructor TMathTokenizer.Create;
begin
  inherited Create;
  FFormatSettings := DefaultFormatSettings;
  if DecimalSeparator = '/' then
    DecimalSeparator := '.';
  Reset;
end;
```

At least one locale in **Windows**, Persian (Iran) (fa_IR), has a slash as decimal separator. Allowing this would cause all sorts of problems with the parser. Does 5/2 represent 5.2 or 2.5? As can be seen the solution adopted was to simply revert to the period as decimal separator. The problem will be exarcebated when a list separator will be required.

It is probably a good idea to let the user change the decimal separator. All that is involved is to change the `DecimalSeparator` (or `FormatSettings.DecimalSeparator`) field to the desired value. The decimal separator cannot be any character, but no error checking is done. However a utility method is included in the tokenizer:

```
      { Returns true if the given character c is an allowable
decimal
        separator. Usually the decimal symbol is a punctuation
mark but
        it cannot be an operator, a digit, a letter nor white
space.}
    function canBeDecimalSeparator(c: char): boolean;
```

There is more to national language support. In particular two functions to convert expressions from a locale dependant form to a “neutral” form are included in the source code. They will be discussed in the next chapter and againt later on.

2.8 Conditional directives

A file named parser.inc, which contains compilation directives, is included in all files that constitute the parser: parsertypes.pas, parserconsts.pas, mdmathtokenizer.pas and mdmathevaluator.pas. The file mostly documents the directives that could be defined by the

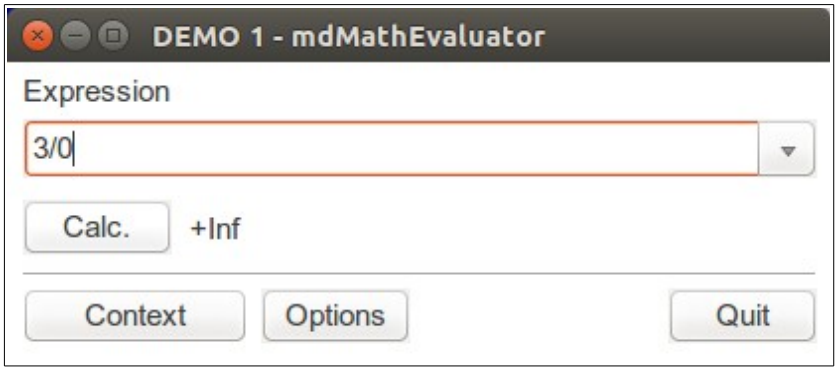
programmer to modify certain aspects of the parser. The most important of these is the `NO_EXCEPTION_MASK` has already been discussed.

The `UNIT_INFO` directive is defined to provide status information in the demonstration program. It will not be defined in the final version of the compiler.

The include file does contain some sanity checks. For example, if a RELEASE directive is defined, then UNIT_INFO will be undefined. More importantly, NO_EXCEPTION_MASK will be undefined if the Gtk2 widget set is used. For that reason, parser.inc should always be included in the parser files even if not directives are defined by default.

2.9 Demonstration Program

A small demonstration program is included with the parser's source code. It contains an edit box in which the expression to be evaluated is entered. Evaluation will be done when the button [Calc.] is pressed. The result is shown to the right of the button. The result is either a numeric value if the expression is correct or an error message or a pop-up dialogue if an exception is raised. For example, the '2+8' displays 10 no matter what the settings of `RaiseExceptions` and what the exception mask is. On the other hand '3/0' displays `+Inf` if division by zero is masked. Since no exception is raised, it does not matter whether `RaiseExceptions` is true or false. Instead `Evaluation error: division by zero` will be displayed if the exceptions are not raised and otherwise the result is shown as blank and exception dialogue box is shown displaying the same error message:



`ExZeroDivide` included	`RaiseException`	Displayed response
`true`	`false`	`+Inf`
`true`	`true`	`+Inf`
`false`	`false`	`valuation error: division by zero`
`false`	`true`	<blank> & exception dialogue box

The demonstration program was compiled and run without problems with **Lazarus** 1.7 and **Free Pascal** 3.0 on **Linux** (with the Gtk2 widget set on **Ubuntu** 14.04.4) and **Windows** 10. It

was also tested with **Lazarus** 1.4.4 and **FPC** 2.6.4 on **Windows** 7. It should work with older versions and with **Delphi** with minimal changes.

The `Options` button opens a dialogue which will let the user set ther parser's decimal symbol, the FPU exception mask, and the parser's raise exception flag. It is not possible to unmask the precision exception. The `Context` button opens a window which displays the current state of parser.

2.10 Testing

Unit tests are provided for both the parser and the tokenizer. An attempt has been made to have the tests function correctly no matter the type of `float` set in ParserTypes.pas. For example, the tests `TestOverflow` and `TestUnderflow` in `TestParser.pas` use the function `sizeof(float)` compared to `sizeof(single), sizeof(double)` etc. to decide which values to use to create a numerical overflow or underflow.

If `float` is not set to `extended`, there can be problems because the expected results of calculations done by the parser are compared with the same calculations performed by the compiler. The latter calculates results using extended floating point representation. Consequently, there will be differences in some results because of the more aggressive rounding when using double or single precision numbers. To avoid this problem, many of the tests use integer values.

The original unit tests were created with Juanco Anez' **DUnit** for **Delphi**. They have been modified to use with **Free Pascal** unit test framework but they continue to use **DUnit** 's `Checkxxx` terminology instead of **Free Pascal**'s native `Assertxxx` procedures.

3 More Mathematical Operations and Comments

It may be useful to allow comments in expressions. Since these are, by definition, ignored by the parser, adding comments only requires a change in the tokenizer. Defining the syntax of comments to ensure they are recognized and then treating them as white space will take care of this addition to the parser.

Three mathematical operations will be added. Integer division and modulo division require minimal changes to the grammar. The most useful addition to the parser, exponents, 12^3 for example, requires substantial changes to the grammar so that the parser will need to be overhauled. An additional, fourth, private function to be called `Exponent` will be added to handle the new syntactic element and some changes to the previous three methods `Term`, `Expression` and `Factor` must be made.

Two other sets of parentheses will be introduced: square brackets '[ ]' and curly brackets '{ }'. This will make it easier to write complicated expressions: `3+(4/(9-(5+8)))` can the be written as `3+[4/(9-{5+8})]`. This "visual candy" is basically a bookkeeping exercise and does not involve meaningful change to the grammar or parser.

The final change continues the handling of format settings but does not require modifying the grammar or parser. Encoding and decoding for expressions as language neutral strings is provided. The routines will convert the decimal separator of numbers to the period '.' no matter what the user's locale is and vice versa.

3.1 Changes to the tokenizer

The operators corresponding to the integer division and modulo division are ':' and '%' respectively. The operator corresponding to exponentiation will be the caret '^'. It is a simple matter to add these to the list of operators to be recognized. Because in some cases '**' is also used to indicate exponentiation, the two-character operator will be defined as a synonym. Two character operators will be discussed later on.

The additional parenthesis are added as an optional element of the grammar. Of course, the allowable set of parenthesis should be set when initializing a program; it hardly makes sense to make this a user option. The following procedures and read only properties are added to the `TMathTokenizer` class:

```
        (* Add character aValue as an accepted left parenthesis. A
matching
           right parenthesis will also be added. Only '{' and '['
can be
           added, all other characters are ignored. Initially these
           are already accepted and do not need to be added. *)
    procedure AddLeftBrace(aValue: char);
```

```
        (* Remove character aValue from the list of accepted left
parenthesis.
           The matching right parenthesis will also be removed.
Only '{' and
           '[' can be removed, '(' is always accepted as a left
parenthesis. *)
    procedure DeleteLeftBrace(aValue: char);

        (* List of accepted left parenthesis. By default these are
'([{'. *)
    property LeftBraces: string read FLeftBraces;

        (* List of accepted right parenthesis. By default these are
')]}'. *)
    property RightBraces: string read FRightBraces;
```

As the comments indicate, by default there are three pairs of parentheses: '()', '[]' and '{}'. Only the second and third pairs can be removed, and added back. This is done with the `AddLeftBrace` and `DeleteLeftBrace` methods that are called with the specific left brace character ('[' and '{'). If any other character, including '(', is provided as a parameter to either function, nothing happens.

In addition to being denoted with the caret '^' , exponentiation is indicated with the double asterisk '**'. Also comments will start with a '//'. The tokenizer must now recognize operators that may be one or two characters long. This is done by introducing the `Peek` method that returns the character following the current token. It is easily implemented because the tokenizer's private field `FNextCharPos` already points to the first character following the current token. The only subtlety involved is the possibility that the end of the source has already been found. In that case, `Peek` will return a `#0` character which is fine because that can never be matched with the second character of an operator.

```
function TMathTokenizer.Peek: char;
begin
  if FNextCharPos > length(FSource) then
    result := #0
  else
    result := FSource[FNextCharPos];
end;
```

When the operator '*' is found, the procedure `GetOperator` uses `Peek` to check if the following characters is also a '*'. If it is not, then a `ttMult` operator was found and that is what is returned. If the second asterisk is found then the MoveByOneChar method is invoked to include the second asterisk in the current token and a ttCaret operator is returned as if the operator is the equivalent '^'.

A similar check is performed when a '/' is found. A peek at the following character is need to decide if this is a single slash which denotes division or a double slash indicating the start of

the comment. In the latter case, the `MoveToEndOfLine` method is invoked to include the rest of the source line into the current token. This task is somewhat complicated by the various sequences of characters that are used on different systems to mark the end of a line. The following routine should work when the end of lines are marked by any combination of a carriage return (CR) and line feed (LF). Initially, all characters that are not end of line markers are accumulated in the token and then all characters that are either a CR or LF are added to the token. This does mean that all empty lines following the comment will be included in the latter.

```
procedure TMathTokenizer.MoveToEndOfLine;
begin
  while (FNextCharPos <= length(FSource))
  and not (FSource[FNextCharPos] in [LF, CR]) do
    MoveByOneChar;
  while (FNextCharPos <= length(FSource))
  and (FSource[FNextCharPos] in [LF, CR]) do
    MoveByOneChar;
end;
```

Eventually, it could be useful to keep track of the line and column count of each token when parsing long mathematical expressions over multiple lines. That way the user could be informed of the line and column where a syntax error is found.

Beside the addition of the `Peek` and `MoveToEndOfLine` methods, most of the change to the tokenizer is thus found in the `GetOperator` procedure:

```
  procedure GetOperator(c: char);
  begin
    MoveByOneChar; {skip c}
    case c of
      '%': result := ttPercent;
      '*': if Peek <> '*' then
             result := ttMult
           else begin
             result := ttCaret;
             MoveByOneChar;
           end;
      '+': result := ttPlus;
      '-': result := ttMinus;
      '/': if Peek <> '/' then
             result := ttDiv
           else begin
             result := ttComment;
             MoveToEndOfLine;
           end;
      ':': result := ttColon;
      '^': result := ttCaret;
    else
```

```
      begin
        if (pos(c, FLeftBraces) > 0) then
          result := ttLeftParenthesis
        else if (pos(c, FRightBraces) > 0) then
          result := ttRightParenthesis
        else
          result := ttUnknown;
      end;
    end;
  end;
```

3.2 Wirth Syntax Notation

Adding exponentiation requires modification of the simple grammar of allowed mathematical expressions. Since the changes to the code are short, this is a good time to introduce a formal syntax notation called Wirth syntax notation (WSN), proposed by Niklaus Wirth in 1977.

Here is an example of a "production", basically a definition using WSN.

```
Digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .
```

Four symbols are used in the above definition. The equals sign ‘=’ indicates a production where the element to the left of the = is defined by the elements to the right. The terminal period ‘.’ denotes the end of the definition. Items enclosed in quotation marks ‘ " ’ are literals. The vertical bar “|” stands for disjunction or. A digit is thus any one character from 0 to 9.

To simplify things, Wirth adds three constructs, curly brackets that denote repetition, square brackets that denote optional elements, and parenthesis that serve to group items.

Repetition is denoted by curly brackets: {a} stands for <none> | a | aa | aaa | ….
Optionality is expressed by square brackets: [a]b stands for ab | b.
Groupings are indicated by parentheses: (a|b)c stands for ac | bc.

Here’s the definition of a number in WSN:

```
Number =
  Digit{Digit} [ "." {Digit} [("E" | "e") ["-" | "+"]
Digit{Digit}]].
```

Here is an alternative definition:

```
Integer       = Digit {Digit} .
SignedInteger = ["-" | "+"] Integer .
Number        = Integer [ "." {Digit} [ ("E" | "e")  SignedInteger]]
.
```

The sequence `Digit {Digit}`, may be slightly unsettling at first blush. It is necessary to specify that a number, or an integer, must start with at least one digit. The rule `{Digit}` does not represent a sequence of digits containing at least one digit since it could be empty.

Using WSN, here is a definition of our simple grammar as developed so far:

```
      Term       = Factor { ("*" | "/" ) Factor } .
      Expression = Term { ("-" | "+") Term } .
      Factor     = Number | "(" Expression ")" | ("-" | "+")
Factor .
```

3.3 Integer Division

The result of a division such as 7 ÷ 3 yields a quotient 2 and a remainder 1 such that the dividend is equal to sum of the remainder and the quotient multiplied by the divisor: 7 = 3*2 + 1. It may be useful to obtain the quotient or the remainder. Two operators will be introduced to return these results. The quotient will be represented by the infix operator ‘:’ , the remainder (modulo division) by the infix operator ‘%’. Hence 7:3 = 2 and 7%3 = 1.

These operations have the same precedence as the normal division. Hence the production of a term which was in Wirth’s syntax notation

```
      Term = Factor { ("*" | "/" ) Factor } .
```

is now

```
      Term = Factor { ("*" | "/" | ":" | "%"  ) Factor } .
```

The usual names for the functions are `div` and `mod`. They are found in Pascal (**Free Pascal** and **Delphi** among others). However, the domain of these functions in Pascal is limited to integers. It was decided to implement the functions instead of using Pascal’s primitive functions. For one, there are a number of definitions and implementations of the functions and it appears as if **Free Pascal** and **Delphi** doe not follow the specified definition in the ISO standard for Pascal. That is probably in itself not a bad thing as the definition found in the Pascal standard has been questioned (Boote (1992) and Leijen (2001)). But it does mean that the result would depend on the compiler used. It would also be nice to be able to obtain a quotient of 2 when 8.2 is divided by 4.1. Especially because the parser does not have a specific integer type, all numbers are real values.

Both Boote and Leijen seem to favour what they call the Euclidean definition of the functions, but testing shows that **Free Pascal** and **Delphi** now implement what both authors call the truncated definition. The table below shows the difference between the two definition. *D* is the dividend or numerator, *d* is the divisor or numerator, *q* the quotient (an integer) and *r* the remainder.

		Euclidean			Truncated		
D	*d*	*q*	*q*×*d*	*r*	*q*	*q*×*d*	*r*
8	3	2	6	2	2	6	2
8	−3	−2	6	2	−2	6	2

−8	3	−3	−9	1	−2	−6	−2
−8	−3	3	−9	1	2	−6	−2

According to the Euclidean definition, $q \times d \leq D$ whatever the signs of D and d. With the truncated definition $q \times d \leq D$ if $D > 0$ and $q \times d \geq D$ if $D < 0$ whatever the sign of d. Of course, in all cases, q and r are such that $D = q \times d + r$. Which is better? While there are mathematical reasons to prefer the Euclidean definition, a programmer may very well prefer the truncated definition which will agree with **Free Pascal** and **Delphi** for integer arguments. In any case, the latter was chosen be default but that behaviour can be modified by defining the compiler directive `EUCLIDEAN_DIV_MOD`. A new unit, minimath.pas, contains the procedures created to handle these new operations.

```
{$IFDEF EUCLIDEAN_DIV_MOD}

function IMod(const X, Y: float): float;
var
  Q: integer;
begin
  Q := trunc(X/Y);
  result := X - Q*Y;
  if result < 0 then begin
    if Y > 0 then begin
      result := result + Y;
    end
    else begin
      result := result - Y;
    end;
  end;
end;

function IDiv(const X, Y: float): Integer;
begin
  result := trunc(X/Y);
  if X - Result*Y < 0 then begin
    if Y > 0 then begin
      dec(result)
    end
    else begin
      Inc(result);
    end;
  end;
end;

{$ELSE}
{Default Borland TP / Delphi div mod}

function IMod(const X, Y: float): float;
begin
```

```
  result := X - trunc(X/Y)*Y;
end;

function IDiv(const X, Y: float): Integer;
begin
  result := trunc(X/Y)
end;

{$ENDIF}
```

3.4 Exponentiation

Introducing exponentiation modifies the grammar to some extent. Here is the new grammar in Wirth's syntax notation:

```
Exponent   = Factor { ("^" | "**") Exponent } .
Term       = Exponent { ("*" | "/" ) Exponent } .
Expression = Term { ("-" | "+") Term } .
Factor     = Number | "(" Expression ")" | ("-" | "+") Factor .
```

There is no change to the definitions of a factor and an expression. A term which was a sequence of factors is now a sequence of exponents. And, of course, the addition of integer quotient and modulo division does make the a term somewhat more complex. It remains, nevertheless, a simple change to make :

```
function TMathEvaluator.Term: float;
begin
  result := Exponent;
  repeat
    if TokenType = ttMult then begin
      NextTrueToken;
      result := result * Exponent;
    end
    else if TokenType = ttDiv then begin
      NextTrueToken;
      result := result / Exponent;
    end
    else if TokenType = ttColon then begin
      NextTrueToken;
      result := IDiv(result, Exponent);
    end
    else if TokenType = ttPercent then begin
      NextTrueToken;
      result := IMod(result, Exponent);
    end
   else begin
      exit;
    end;
  until false;
end;
```

While the expected definition of an exponent could have been

```
Exponent = Factor { ("^" | "**") Factor }
```

it was not chosen. The reason is that, according to that definition, 4^3^2 would yield (4^3)^2 = 4 096. Using the usual mathematical notation leads to a different answer

$$4^{3^2} = 4^9 = 262\,144 \neq (4^3)^2.$$

This is really a personal choice, because both definitions are used: in **Excel**, 4^3^2 = 4 096 and in **Perl** 4**3**2 yields 262 144. Here is the function for exponents. It's the simplest of the functions implementing the syntactic elements.

```
function TMathEvaluator.Exponent: float;
begin
  result := Factor;
  while TokenType = ttCaret do begin
    NextTrueToken;
    {$IFDEF EXPONENT_LEFT_ASSOCIATIVE}
    result := Power(result, factor)
    {$ELSE}
    result := Power(result, Exponent());  // don't forget () in
fpc mode
    {$ENDIF}
  end;
end;
```

By default, the directive `EXPONENT_LEFT_ASSOCIATIVE` is not defined. If it were defined then **Excel** like behaviour, with a left associative exponentiation operator would be obtained.

3.5 Precedence

There is another ambiguity that needs to be addresses: the meaning of the expression `-3^2`. Written in standard mathematical notation -3^2 is usually interpreted as meaning $-(3^2) = -9$. However the parser will actually yield 9 as a result. This is because the unary − has the highest precedence. Looking at the grammar, the unary minus is handled in the `Factor` routine.

Operator	Operation	Priority
()	parenthesis	5
-	negation	4
^ (or **)	exponentiation	3

*	multiplication	2
/	division	2
:	integer division	2
%	modulo	2
+	addition	1
−	subtraction	1

Operators with higher priority are done first.

Except for exponentiation, when operations of the same priority are done in succession, they are grouped left to right. Hence 2+3−1 is actually computed as (2+3)−1 and 2−3+1 is computed as (2−3)+1. Similarly 8*2/4 is interpreted as (8*2)/4 = 4 and 8/2*4 is calculated as (8/2)*4 = 16.

It is best to use parentheses to control the precedence of operations when in doubt.

3.6 Universal Conversion of Expressions

When taking into account the user's locale and allowing the user to change format settings, it becomes a bit more complicated to exchange data via text files or the clipboard. One solution is to convert all expressions into a common language for exchange. That's what is behind the pretensious sounding title of this last section.

The choice of the common language is arbitrary, of course. Indeed an artificial language was used in an earlier version of the conversion routines. This is no longer the case because it made difficult reading any expression saved to a text file. Instead, the constant use of the period as decimal among English language regions is adopted.

The tokenizer provides two utility functions:

```
    function Encode(const source: string): string;
    function Decode(const source: string): string;
```

The first function `Encode` returns the source string converting all current decimal separators to periods in numbers. Conversely, `Decode` returns the source string converting all periods to the current decimal separator in numbers. Note that the "current decimal separator" refers to the value of the `DecimalSeparator` (actually `FormatSettings.DecimalSeparator`) field not the value of `DefaultFormatSettings.DecimalSeparator`.

Only decimal separators appearing in numbers should be converted. Hence conversion cannot be a simple search and replace. It is necessary to scan the source and identify the numeric sub strings and then convert any contained decimal separator. Might as well use a `TMathTokenizer` to do this:

```
function TMathTokenizer.Encode(const source: string): string;
var
  tempTokenizer: TMathTokenizer;
  s: string;
  p: integer;
begin
  result := '';
  tempTokenizer := TMathTokenizer.create;
  try
    tempTokenizer.FormatSettings := FormatSettings;
    tempTokenizer.Source := source;
    while tempTokenizer.NextToken <> ttEOS do begin
      s := tempTokenizer.Token;
      if tempTokenizer.TokenType = ttNumber then begin
        p := pos(DecimalSeparator, s);
        if p > 0 then
          s[p] := EnFormatSettings.DecimalSeparator;
      end;
      result := result + s;
    end;
  finally
    tempTokenizer.free;
  end;
end;
```

Note how the format settings of the temporary tokenizer are set to the current tokenizer's settings before parsing. Then its simply a matter of scanning each token and adding it to the result string, converting the decimal separator when a number is found. The substituted decimal separator is taken from a constant record `EnFormatSettings` that contains the **Free Pascal** default format settings (which correspond to the en_US locale). That record is defined in the interface portion of `mdMathTokenizer.pas` file.

The `Decode` function works pretty much the same way with the difference that the temporary tokenizer's format settings are copied from `EnFormatSettings` and the substituted decimal separator is the current decimals separator.

Some may not need all this. In that case a directive called `NO_NLS` can be defined so that the functions `Encode` and `Decode` merely copy the source without any change. It is probably best to nevertheless use these functions event when the directive `NO_NLS` is defined. Adding national language support later will then not require change to the application's source.

4 Adding Constants, Functions and Variables

Scientific calculators tend to have a plethora of built-in functions which usually include the trigonometric functions (sin, cos, tan, etc.), hyperbolic functions (sinh, cosh, tanh, etc.) , their inverses, and other mathematical functions (abs, sqrt, ln etc.). Often constants, such as *pi* and Euler's number *e*, have a dedicated key. Perhaps many more are included especially if the calculator is used by scientists.

Adding such functions and constants as well as user defined variables is the subject of this third article on parsing and evaluating mathematical expressions. However, there is no question of adding buttons here. Functions are *named* calculations, constants and variables are *named* numbers. The parser will be modified to recognize these names and to call the routine that performs the calculation or retrieves the number.

All built-in constant, functions and user defined variables are descendants of an abstract class called `TTarget`. The name was chosen because most descendant classes will be the targets of the `Call` method of the parser. The addition of built-in functions and constants requires a small change to the tokenizer so that it will recognize the names of the functions and return the appropriate target. It also requires the addition of a unit that will contain the definitions of the built-in functions. Finally, the grammar and parser must be modified to allow calling these targets. These changes are considered in turn.

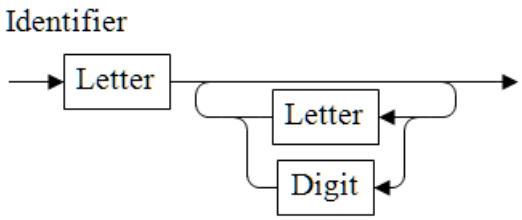
4.1 Identifiers

To name functions, a new type of syntactic element called an identifier must be introduced. An identifier is any sequence of letters and digits that begins with a letter. Here is one possible formal definition using Wirth's syntax notation:

```
Digit      = "0" | ... | "9" .
Letter     = "a" | ... | "z"|"A" | ... | "Z" .
Identifier = Letter { Letter | Digit } .
```

Some languages include the underscore `"_"` as a letter and some allow the period `"."` to be included in an identifier as long as it is not the first character.

Since no other token type begins with a letter, lexical analysis of the source remains simple. The tokenizer can still use the first letter to identify the token type and then accumulate the subsequent letters and digits.



Only a couple of lines need to be added to the `NextToken` method. After it is determined that the first character of a token is not a space, nor a digit (which would be the start of a number) a test is made to see if it is a letter. In the affirmative, the letter marks the beginning of an identifier which is fetched with the `GetIdentifier` procedure.

```
var
  c: char;
begin
  // move past current token
  FTokenStart := FNextCharPos;
  FTokenLength := 0;
  FTarget := nil;
  if FNextCharPos > length(FSource) then
    result := ttEOS
  else begin
    c := FSource[FTokenStart];
    if isWhite(c) then
      GetWhite
    else if isDigit(c) then
      GetNumber
    else if isLetter(c) then
      GetIdentifier
    else
      GetOperator(c);
  end;
  FTokenType := result;
end;
```

Note that the routine starts by assuming that the token about to be found is not a target. If the token starts with a letter then the `GetIdentifier` routine is called.

```
  procedure GetIdentifier;
  begin
    result := ttIdentifier;
    MoveByOneChar;
    while (FNextCharPos <= length(FSource))
    and isAlphaNumeric(FSource[FNextCharPos]) do
      MoveByOneChar;
    if not InternalTargets.Find(Token, FTarget) then begin
      FTarget := nil;
      result := ttUnknownIdentifier;
    end
    else if (FTarget is TKeyword) then begin
      result := TMathTokenType(TKeyword(FTarget).Id);
      FTarget := nil;
    end;
  end;
```

There is not much to the procedure. It starts by setting `NextToken's` result to `ttIdentifier` and then accumulates all subsequent letters and digits into the token. Then it

checks if the identifier, stored in the `Token` field, is contained in the list of internal targets. If the identifier is not found, `NextToken's` result is changed to `ttUnknownIdentifier`, letting the parser know that an error has occurred. If the identifier is in the list, the corresponding target is returned in the `FTarget` variable. A test is then made for a target of type `TKeyword`. These correspond to the three identifiers `mod` , `div` and `def`. If that is the case, `NextToken's` result is changed to the corresponding token type `ttPercent`, `ttColon`. or `ttDefine`.

The two `nil` assignment to `FTarget` in case of an error are probably manifestations of paranoia; `FTarget` should already be `nil`.

As can be seen, there really is not much involved in lexical analysis of identifiers.

4.2 Built-in Functions

All named functions, variables, constants, and keywords are descendants of the abstract class `TTarget`.

```
TTarget = class
 private
   FCount: integer;
 protected
   procedure SetCount(value: integer); virtual;
 public
   function Calc: float; virtual; abstract;
   property Count: integer read FCount write SetCount;
 end;
```

A target does not know its name. The abstract method `Calc` will be called each time the value of a function or constant is needed. There is only one property, `Count` which is the number of parameters or arguments of the target. For some functions, such as the `avg` (average) or `var` (variance), there is an indeterminate number of variables but for most functions `Count` is a fixed value, usually 1. For that reason the virtual method `SetCount` in `TTarget` does nothing. But in debug mode it does raise an assert since this value should never be set by a descendant of `TTarget`:

```
procedure TTarget.SetCount(value: integer);
begin
  {$IFDEF DEBUG}
  assert(false, 'Cannot set Count');
  {$ENDIF}
end;
```

The simplest descendants of `TTarget` are constants, whose count will be equal to 0:

```
type
  TConstant = class(TTarget)
```

```
  private
    FValue: float;
  public
    constructor create(value: float);
    function Calc: float; override;
  end;

constructor TConstant.create(value: Extended);
begin
  inherited Create;
  FValue := value;
end;

function TConstant.Calc: float;
begin
  result := FValue;
end;
```

The constructor stores the given value in the private field `FValue` and the calc method merely returns the value. Creating the target *e* = 2,718... is quite simple:

```
  TConstant.create(exp(1));
```

If a property permits setting the value we now have a variable:

```
  TVariable = class(TConstant)
  public
    property Value: float read FValue write FValue;
  end;
```

Creating a variable with an initial value of 5 and then changing it value to 10 would look like this:

```
var
  target: TVariable;
begin
  target := TVariable.create(5);
  target.Value := 10;
```

Because the tokenizer searches the list of targets when it finds an identifier, it was decided to create keyword targets which are just as simple as constants and variables.

```
type
  TKeyword = class(TTarget)
  public
    constructor create(anId: integer);
    property Id: integer read FCount;  // TTokenType of operator
    function Calc: float; override;
  end;

constructor TKeyword.create(anId: integer);
begin
  FCount := anId;
```

```
end;

function TKeyword.Calc: float;
begin
  {$IFDEF DEBUG}
  assert(false, 'Cannot calculate a keyword operator');
  {$ENDIF}
end;
```

There's a little bit of cheating in the case keywords `div` and `mod`. Their Calc function is never invoked and `count` field is used to store an integer which will be the ordinal value of the `div` or `mod` operator token type. Hence the creation of the two keyword targets will be

```
TKeyword.Create(ord(ttColon));     // the div (:) operator
TKeyword.Create(ord(ttPercent));   // the mod (%) operator
```

All built-in functions with a fixed number of arguments are descendants of the abstract class `TFunction`.

```
type
  TFunction = class(TTarget)
  public
    Args: dFloatArray;
    constructor create(aCount: integer);
  end;
```

When created, the specified number of expected arguments, `aCount`, is copied into the internal field `FCount` and cannot be changed. The propery `Args` is a dynamic array of floats; the type `dFloatArray` is defined in ParserTypes.pas:

```
type
  float = extended;
  dFloatArray = array of float;
```

The array will contain the numeric values of the function's parameters. As will be discussed later, an implementation of a `TFunction` does not allocate the `Args` field. That must be done by the calling procedure in order to ensure that functions can be called recursively.

Here is the implementation of the log function.

```
type
  _log = class(TFunction)
  public
    function Calc: float; override;
  end;

function _log.Calc: float;
begin
  result := logn(Args[1], Args[0]);
end;
```

When creating this function, the constructor will be called with `aCount = 2`:

```
_log.create(2);
```

Most built-in functions are as simple. By default, the name of all built-in functions and constants begins with an underscore instead of the traditional T. Built-in functions and constants are all defined in the implementation part of the unit `Targets.pas`. None of the details concerning built-in functions need to be exposed. When looking at that unit which is lengthy, it is useful to remember the simple convention used. If the identifier for a function is '`foo`', the associated class is called `_foo`.

Finally there remains the functions with a variable number of arguments. If the maximum number of arguments is not known, the minimum number of arguments is known and used when parsing to ensure that a call to such a function is possible.

```
type
  TVariableFunction = class(TFunction)
  private
    FMinCount: integer;
  protected
    procedure SetCount(value: integer); override;
  public
    constructor create(aMinCount: integer);
    property MinCount: integer read FMinCount;
  end;

constructor TVariableFunction.Create(aMinCount: integer);
begin
  inherited Create(0);  //FCount := 0
  {$IFDEF DEBUG}
  assert(aMinCount > 0, 'Minimum number of arguments less than 1');
  {$ENDIF}
  FMinCount := aMinCount;
end;

procedure TVariableFunction.SetCount(value: integer);
begin
  if Count <> value then begin
    {$IFDEF DEBUG}
    assert(value >= MinCount, 'Too few parameters');
    {$ENDIF}
    FCount := value;
  end;
end;
```

The extra field in this class called `MinCount` is set in the class' constructor. The class also overrides the `SetCount` method and allows the caller to specify the number of arguments that were found. Here is the implementation of the average function which shows how `Count` is used.

```
function _avg.Calc: float;
var
```

```
  i: integer;
begin
 result := 0;
  for i := 0 to Count-1 do
    result := result + args[i]/count;
end;
```

All built-in functions have to be registered, as it were, so that the parser knows their name and the target in order to get its value by invoking its `Calc` method. This is a simple matter of storing the names of the functions in a string list along with the corresponding `TTarget` implementation as object.

```
TTargetList = class
  private
    FList: TStrings;
    function GetCaseSensitive: boolean;
    function GetCount: integer;
    procedure SetCaseSensitive(value: boolean);
    function GetTarget(index: integer): TTarget;
  public
    constructor Create;
    destructor Destroy; override;
    procedure Add(const aName: string; target: TTarget);
    procedure Clear;
    function Delete(const aName: string): boolean;
    function Find(const aName: string; out target: TTarget):
boolean;
    function HasTarget(const aName: string): boolean;
    function NameOf(target: TTarget): string;
    property CaseSensitive: boolean read GetCaseSensitive write
SetCaseSensitive;
    property Count: integer read GetCount;
    property Names: TStrings read FList;
    property Targets[index: integer]: TTarget read GetTarget;
default;
  end;

var
  InternalTargets: TTargetList;
```

The most used methods of this class are `Add` and `Find`.

An internal function is registered by calling the add method with the functions' name and it's instantiated class. Adding the `_log` function defined above is done with the following statement:

```
  InternalTargets.Add(SC_log, _log.create(2));
```

where `SC_log` is a string literal defined equal to 'log' in ParserConsts.pas. If an error occurs, because a function with the given name is already in the list, the target is freed and an exception is raised.

Use of the list is straight forward. When the tokenizer finds an identifier, it searches for it with the method `InternalTargets.Find`. If the identifier is found, `Find` will return true and the matching function or constant in the variable `target`. Otherwise it returns false. The tokenizer returns the result as a token type: `ttIdentifier` or `ttUnknownIdentifier`. It is up to the parser to raise an exception in that latter case. How it handles found internal functions and constants is the subject of the next section.

By default, the target names are not case sensitive. The `InternalTargets.Find` will return true if called with the ‘sin’, ‘sIn’ or ‘SIN’. If the `CaseSensitive` property is set to true, then only lower case identifiers can match a name in the list of internal targets. It is probably best that the programmer set the value of this property at program start-up according to the application’s needs and not present this as an option to the user.

While the usual functions are added to `InternalTargets` in `Targets`’ initialization code, it is possible to add or delete a function to the list at any time. The demonstration program shows how to do so. Programmers should always use a lower case name to a function added to the list of internal targets. Otherwise, the `CaseSensitive` property will not function properly.

Adding a function to the internal list is done by a programmer, it is not the same as having the user add a function that he or she defines at run time. That complication is considered in a later article. However adding user defined constants or variables is relatively simple.

4.3 Parsing a Built-in Function

Two productions need to be added to the grammar, to handle built-in functions and constants. First the optional list of arguments of a function, `Params`, is defined as follows in Wirth syntax notation :

```
Params = "(" Expression { ListChar  Expression} ")"
```

where `ListChar` is a locale dependant list separator. In English language systems this is usually the comma, in French language systems it is the semi-colon. A call to a function is a statement of the following form

```
Call = Identifier [ Params ]
```

A call to a function without any parameters is actually a call to a constant. A call is equivalent to a number. While it may require very convoluted calculations, eventually a called function must resolve into a single number and the call itself can occur only in a factor:

```
Factor = Number | "(" Expression ")" | Call | ("-" | "+") Factor .
```

So the only method of the parser that needs to be changed is `Factor`.

```
begin
  result := 0;
  if TokenType = ttNumber then begin
    result := sfloat(Token);
    NextTrueToken;
  end
  else if TokenType = ttIdentifier then begin
    result := Call;
  end
  else if TokenType = ttPlus then begin
...
```

The function `Call` must take care of parsing the argument list, if there is one, and evaluate the function and return its value. This is easily the most complicated routine in the whole parser. To simplify, the work is divided into two parts. Parsing the argument list is done by another function called `Params`. That method must handle an unknown number of arguments. So it accumulates expressions separated by the `ListChar` until a right parenthesis is found and return the number of arguments found as well as their values. Here is its code.

```
function TMathEvaluator.Params(out FormalArgs: dFloatArray):
integer;
var
  aValue: float;
begin
  if (NextTrueToken <> ttLeftParenthesis) or (Token <> '(') then
    ParserError(meExpectedLeftPar);
  NextTrueToken;
  result := 0;
  setlength(FormalArgs, 0);
  repeat
    avalue := Expression;
    setlength(FormalArgs, result+1);
    FormalArgs[result] := avalue;
    inc(result);
    if TokenType = ttRightParenthesis then begin
      // only () parenthesis allowed for function calls
      if Token <> ')' then
        ParserError(meExpectedRightPar);
      exit;
    end;
    if TokenType <> ttSeparator then
      ParserError(meExpectedSeparator);
    NextTrueToken; // skip ','
  until false;
end;
```

The function returns the number of found expressions and their values are also returned in a dynamic array[2] which grows as expressions are found. Note that in this case only classic

2 The previous version of the parser was meant to be compatible with **Delphi 2**, where dynamic arrays were

parentheses ‘( )’ are allowed.

It is now possible to examine the function `Call`.

```
function TMathEvaluator.Call: float;
var
  ArgCount: integer;
  LTarget: TTarget;
  LArgs: dFloatArray;
begin
  LTarget := Target;
  if LTarget is TFunction then begin
    ArgCount := Params(LArgs);
    TFunction(LTarget).Args := LArgs;
    if LTarget is TVariableFunction then begin
      if ArgCount < TVariableFunction(LTarget).MinCount then
        ParserError(meTooFewParams);
      LTarget.Count := ArgCount
    end
    else if ArgCount < LTarget.Count then
      ParserError(meTooFewParams)
    else if ArgCount > LTarget.Count then
      ParserError(meTooManyParams);
  end;
  result := LTarget.Calc;
  NextTrueToken; // skip ')'
end;
```

The first task of `Call` is to copy the found target held in the field `FTarget` to a local variable named `LTarget`. Then if a check is made to see if the target is a function and in that case Params is invoked with a local dynamic array `LArgs` to get the number of parameters found. Say the original expression was `max(sin(pi), cos(2*pi))`. While parsing `max`, `Params` will find a new target, `sin`, which means the tokenizer will change the value of its `Target` field. That explains why the local `LTarget` variable is used.

A similar argument explains why a local arguments array is used. Recursion would be impossible without it. Imagine parsing `log(log(9, 2), 4)`. The correct answer is approximately 0.832:

log(9, 2) = 3.1699
log(3.1699, 4) = 0.832.

Assume that `LArgs` was not used so that the code would be

```
  LTarget := Target;
  if LTarget is TFunction then begin
```

not available. A `TList` temporarily stored information about the parameters, and their values was returned in a fixed length array explicitly allocated on the heap. That is the most notable difference in the two versions of the parser aside from the treatment of exceptions.

```
    ArgCount := Params(LTFunction(LTarget).Args);
...
```

Then parsing the outer call to log should fill `_log.Args` with [??, 4], where ?? will be replaced by the value of log(9, 2). But when parsing log(9, 2), `_log.Args` would overwritten with [9, 2]. The `_log.Calc` will return 3.1699 which is correct and would be stored at the start of the `_log.Args` array but the original second argument 4 is replaced with the 2. Hence the outer `_log.Calc` would operate on the argument array [3.1699, 2] not [3.1699, 4].

Once `Params` has been invoked and the parameters have been found, `Call` ensure that the number of parameters found is correct. If the target is a function with a variable number of arguments, `Call` checks that at least the minimum number of arguments have been found and if so, sets the count of arguments in the target. If the target is a function with a fixed number of arguments, `Call` checks that the number of arguments found is correct. Finally, `Call` returns the value returned by the target's `Calc` function and skips the trailing right parenthesis.

4.4 List Separator

As mentioned above, adding functions with multiple parameters requires a new operator, the list separator. In accordance with the decision concerning the decimal symbol, the symbol to separate arguments of a function is locale dependant. Actually, since handing the decimal separator involved copying the format settings, it would appear that very little additional work is required to handle the list separator.

```
    property ListSeparator: char read FFormatSettings.ListSeparator
                                write
FFormatSettings.ListSeparator;
```

That is not quite all. There are platform differences with regard to the list separator, not least of which is the unfortunate fact that **Linux** does not define one. Consequently, the list separator on **Linux** systems will be the comma no matter what the decimal separator is. On top of that at least six locales (in **Windows** and perhaps other platforms) use the comma as decimal and list separator. This is an unfortunate state of affairs which is mitigated with the following code in the tokenizer's constructor:

```
  ...
  FFormatSettings := DefaultFormatSettings;
  If (DecimalSeparator = ',') and (ListSeparator = ',') then
    ListSeparator := ';';
  if DecimalSeparator = '/' then begin
    DecimalSeparator := '.';
    ListSeparator := ',';
  end;
```

```
  ...
```

This is discussed in more detail elsewhere (Deslierres 2016b).

The encoding and decoding functions must be modified to handle the list separator. This involves adding a test for the separator in the main loop in `TMathTokenizer.Encode`:

```
    while tempTokenizer.NextToken <> ttEOS do begin
      s := tempTokenizer.Token;
      if tempTokenizer.TokenType = ttSeparator then
        s := EnFormatSettings.ListSeparator
      else if tempTokenizer.TokenType = ttNumber then begin
        p := pos(DecimalSeparator, s);
        if p > 0 then
          s[p] := EnFormatSettings.DecimalSeparator;
      end;
      result := result + s;
    end;
```

When a list separator token is encountered, it is replaced with a comma which is the list separator in English locales. Conversely, in `Decode`, the list separator (the comma) is replaced with the current list separator.

4.5 Missing Mathematical Functions

The `math` unit supplied with **Free Pascal** does not contain all the function needed to implement the built-in functions included in Targets.pas. Consequently a number of functions have been added to the minimath.pas unit:

```
function Sinh(const X: float): float;
function Cosh(const X: float): float;
function Tanh(const X: float): float;
function CotH(const X: float): float;
function SecH(const X: float): float;
function CscH(const X: float): float;
function ArcSecH(const X: float): float;
function ArcCotH(const X: float): float;
function ArcCscH(const X: float): float;
```

Because **Free Pascal**'s math.pas does not contain an inverse hyperbolic cosecant function and minimath's implementation is correct, there is not need for `FIX_ARCCSCH` directive in `Parser.inc` which was needed in the **Delphi** version of the parser,

4.6 Variables

While performing a series of calculations, the user could find it useful to reuse the last calculated value in a current expression. And since this is an easy feature to implement, we will make it possible to get at the second to last calculated value also.

Two local fields are added to the parser to store these results:

```
  TMathEvaluator = class(TMathTokenizer)
  private
    FLastResult: float;
    F2ndLastResult: float;
   ...
```

Note how inclusion is controlled by the `LAST_RESULTS` directive defined in Parser.inc. These values are updated each time an expression is correctly calculated:

```
function TMathEvaluator.EvaluateExpression(const aSource: string):
float;
begin
  Source := aSource;
  NextTrueToken;
  try
    result := Expression;
    if (TokenType <> ttEOS) then
      ParserError(meExpectedEOS);
    F2ndLastResult := FLastResult;
    FLastResult := result;
  except
  ...
```

There remains changes to the grammar to access these variables. Two operators are added: '$' which will denote the last calculated result and '$$' which will represent the second do last results. That means that two items are added to the `TMathTokenType` enumeration in ParserTypes.pas: and the tokenizer must be adjusted:

```
type
  TMathTokenType = (
    ...
    {: Values returned by NextToken or NextTrueToken when the
operators
       "$" and "$$" are found.}
    ,ttDollar, ttDblDollar
    ...
```

```
procedure GetOperator(c: char);
  begin
    MoveByOneChar; {skip c}
    case c of
      '$': if Peek <> '$' then
             result := ttDollar
           else begin
             result := ttDblDollar;
             MoveByOneChar;
           end;
  ...
```

A small change to the definition of a factor is required:

```
Factor = Number | "(" Expression ")" | Call | ("-" | "+") Factor|
"$" | "$$".
```

which means that the Factor function must now check for these tokens and return the appropriate value when they are found:

```
  ...
  else if TokenType = ttDolar then begin
    result := FLastResult;
    NextTrueToken; // skip '$'
  end
  else if TokenType = ttDblDollar then begin
    result := F2ndLastResult;
    NextTrueToken; // skip '$$'
  end
  else if TokenType = ttEOS then
  ...
```

That is all. From now on, expression such as `'$'`, `'10*$$'` and `'sin($/$$)'` are possible. There is good reason to add a boolean property to control the availability of this feature allowing for run-time tweaking. This is done with the `KeepLastResults` boolean property which is included in the source code but not shown here.

While these additions may be useful, it would probably be even more useful if the user were able to store a value under a name in order to use it in later expressions. I added this functionality in later versions of **Caleçon** by including a second list of targets called `ExternalTargets`. Keeping these additional targets separate from the built-in functions and constants make is easy to delete them all should the user want that.

This feature is not difficult to implement. Again a directive, `NO_EXTERNALS`, can be defined to disable this feature. Most of the change is done in Targets.pas where a new type of list of targets is defined:

```
  TExternalTargetList = class(TTargetList)
  private
    FinternalList: TTargetList;
  public
    procedure Add(const aName: string; target: TTarget); override;
    property InternalList: TTargetList read FInternalList write
FInternalList;
  end;
```

Its additional property `InternalList` is meant to point to the list of internal targets, but by default is nil. The `Add` procedure override the procedure by the same name in `TTargetList` (which must be made `Virtual`) and checks if the identifier `aName` is already used by an internal function or constant. Of course if `InternalList` is `nil`, this check cannot be done

and that means that it will be possible to create a variable which will “hide” a built-in function or constant.

```
procedure TExternalTargetList.Add(const aName: string; target:
TTarget);
var
  FoundTarget: TTarget;
  E: EParserError;
begin
  if assigned(FInternalList)
  and FInternalList.Find(aName, FoundTarget) then begin
    target.free;
    E := EParserError.Create(SmeIdentIsBuiltInName);
    E.Error := meIdentIsBuiltInName;
    E.TokenStart := 0;
    E.TokenLength := 0;
    raise E;
  end;
  inherited Add(aName, target);
end;
```

Note that a new `EParserError`, `meIdentIsBuiltInName` was added in ParserTypes.pas along with the corresponding error message in ParserConsts.pas. The unit’s initialization and finalization code takes care of creating the external target list much like it looked after the list of internal targets:

```
initialization
  InternalTargets := TTargetList.create;
  LoadInternalTargets;
  ExternalTargets := TExternalTargetList.Create;
  ExternalTargets.InternalList := InternalTargets;
finalization
  ExternalTargets.Free;
  InternalTargets.Free;
end.
```

Once a variable has been defined and added to the list of external targets, the user can use it in expressions just like using an internal constant. This requires no change to the grammar, and only a small change to the tokenizer `GetIdentifier` routine:

```
  procedure GetIdentifier;
  begin
    result := ttIdentifier;
    MoveByOneChar;
    while (FNextCharPos <= length(FSource))
    and isAlphaNumeric(FSource[FNextCharPos]) do
      MoveByOneChar;
    if ExternalTargets.Find(Token, FTarget) then
      exit;
    if not InternalTargets.Find(Token, FTarget) then begin
   ...
```

When an identifier is found, the list of external targets is searched first and, only if the identifier is not found, is the list of internal targets searched. The order is important. It explains why (if `ExternalTargets.InternalList` is set to `nil`) an added variable can hide a built-in function.

4.7 Target Hint

While the parser described above can be considered complete, it is useful to add a description or hint property for each target. This is a simple matter:

```
Type
  TTarget = class
  private
    FCount: integer;
  protected
    FHint: string;
    procedure SetCount(value: integer); virtual;
  public
    function Calc: float; virtual; abstract;
    property Count: integer read FCount write SetCount;
    property Hint: string read FHint write FHint;
  end;
```

Again this is an optional element controlled by the conditional directive `TARGET_HINT` which is defined by default in Parser.inc. Hints are set when the target is added to the list of targets:

```
 TTargetList = class
  public
    ...
    procedure Add(const aName: string; target: TTarget;
                  const aHint: string = '');  virtual;
```

The change to the procedure is minimal, the addition of a single line of code:

```
procedure TTargetList.Add(const aName: string; target: TTarget;
                          const aHint: string = '');
begin
  target.Hint := aHint;
  try
    FList.AddObject(aName, target);
  except
    target.Free;
    Raise;
  end;

end;
```

The `Add` procedure of the `TExternalTargetList` class also needs to be adjusted. It merely involves passing on the hint to `TTargetList.Add`. The hints are string literals found in

ParserConsts.pas. The only other change needed is in the `LoadInternalTargets`:

```
procedure LoadInternalTargets;
begin
  InternalTargets.Add(SC_sin, _sin.create(1), S_sin_Hint);
  InternalTargets.Add(SC_cos, _cos.create(1), S_cos_Hint);
  ...
```

The demonstration program included with the parser has been updated to reflect the additions discussed above. Because it is now a program of some complexity, another example application is provided. Called **tcalc** it is a command line calculator. It will print the result calculated from an expression or some help:

```
Usage:
  tcalc ( --Help | (-h | -?) identifier | "expression" )
```

So `tcalc -h sin` will print out the following

```
sin Returns the sine
It is a one parameter function
```

while `tcalc --Help` will print out the list of all targets names and their hint. Typical use is as follows

```
michel@hp:~/tcalc$ ./tcalc "sin(2/3)"
0,61836980306973701
```

Don't forget to quote the expression to avoid problems with the bash interpreter.

Here is the complete source code for the application

```
program tcalc;

{$mode objfpc}

uses
  {$IFDEF UNIX}
    clocale, // see mdMathTokenizer
  {$ENDIF}
  SysUtils, ParserTypes, Targets, mdMathEvaluator;

var
  Source: string;
  Evaluator: TMathEvaluator;

procedure GetSource;
...

var
  res: float;
begin
```

```
  GetSource;
  Evaluator := TMathEvaluator.create;
  try
    Evaluator.ExceptionMask := [exPrecision];
    res := Evaluator.EvaluateExpression(Source);
    if Evaluator.ErrorCode = meNoError then
      writeln(Format('%g', [res]))
    else begin
      write(Evaluator.ErrorMessage);
      if (Evaluator.ErrorCode in ParserErrors)
      and (Evaluator.TokenLength > 0) then
        write(' found "', Evaluator.Token, '"');
      writeln;
    end;
  finally
    Evaluator.free;
  end;
end.
```

`GetSource` scans the command line, takes care of displaying usage or help if required and otherwise fills the `Source` variable. In that case, a `TMathEvaluator` is created. All floating point exceptions, except for `exPrecision` are enabled, but the class will not raise exceptions (this is the default behaviour). The expression found on the command line is then passed on to the `EvaluateExpression` function with the result stored in the variable `res`. Since exceptions are not raised, the evaluator's `ErrorCode` property is checked. If no error is found, the program prints the saved result. Otherwise it prints the error message. The class is then freed as the program is terminated.

Hopefully, this example shows how simple it is to use the parser. The next section discusses the parser used in **Caleçon** which corresponds to the parser described so far except for more elaborate national language support.

5 Extended National Language Support

Most important applications are localized meaning that there are multiple versions adapted to different languages. Some applications go beyond translating the user interface. Function names in formulas in **LibreOffice.Calc** and **Excel** are also translated. Thus in the English version the function that returns the square root of a number is called `SQRT` while in the French version it is called `RACINE`. When a spreadsheet, created and saved with a localized French version, is opened in another language version of that application, all formulas are correctly translated. Clearly, the spreadsheet is stored in a language neutral fashion. Mimicking these two behaviours is the subject of this last section on the one pass parser. Of course, it must be possible to translate target hints just as easily as their name.

Since this is the last version of the one pass parser, the units are wrapped up in a package.

5.1 Localization in Free Pascal

The subject has been broached in section 2.7 User Locale (p. 22) and 3.6 Universal Conversion of Expressions (p. 35) which were concerned with format settings. Now translations of strings will be examined.

Free Pascal and **Lazarus** implement the **GNU gettext** framework for message translation. And it does so in a seamless way compared with **dxgettext** in **Delphi**. There is a caveat if you are used to the former and now starting to using **Free Pascal**. Whereas the **dxgettext** recommends not using resource strings if they are to be translated, **Free Pascal** expects that they are used exclusively. It is also better to avoid the tools supplied by **dxgettext** to extract messages from sources and so on. Some do not yet support features of **GNU gettext**, such as the message context, that are used in **Free Pascal**.[3]

As mentioned in the previous paragraph, the key to translating the names and hints of the built-in functions is to include them in the application as resource strings. Hence all the target names (in English) should be added to ParserConsts.pas as resource strings:

```
  {$IFDEF NO_NLS2}
const
  {$ELSE}
resourcestring
  {$ENDIF}
    S_abs = 'abs';
    S_acos = 'acos';
    S_acosh = 'acosh';
    ...
    S_varp = 'varp';
```

Only a few of the names are shown above. Note how the names are declared as resource

3 Messages in the dxgettext group on yahoo in July 2016 indicate that this shortcoming is being addressed. Nevertheless, I see no real need to install **dxgettext** if **Lazarus** and **Free Pascal** are used exclusively.

strings if the `NO_NLS2` directive is not defined or as literal constants if the directive is defined. Built-in targets are added to the list of internal targets in the following fashion:

```
procedure LoadInternalTargets;
begin
  InternalTargets.Clear;
  InternalTargets.Add(S_sin, _sin.create(1), S_sin_Hint);
  InternalTargets.Add(S_cos, _cos.create(1), S_cos_Hint);
  ...
  InternalTargets.Add(S_varp, _varp.create(1), S_varp_Hint);
```

Assume that `NO_NLS2` is not defined and that the user loads a new language file (*.po). Then all the resource strings will be translated. A call to `LoadInternalTargets` will clear the list, and then all the targets will be added back into the list with their translated names. Pretty much the same thing happens when a program is started. The resource strings are translated to the user's language as defined by the system's locale and then `LoadInternalTargets` is executed in the Targets.pas initialization code.

If only it were that simple.

5.2 Universal Translations of Expressions

Encoding and decoding of expression must now translate target names. Consequently a target will now contain its English name even as it does not know its name in the user's national language.

```
  TTarget = class
  private
     FCount: integer;
   protected
     FHint: string;
     FInternalName: string;
     procedure SetCount(value: integer); virtual;
   public
     function Calc: float; virtual; abstract;
     property Count: integer read FCount write SetCount;
     property Hint: string read FHint write FHint;
     property InternalName: string read FInternalName write
FInternalName;
   end;
```

Adding a target to a `TTargetList` requires specifying the English name (called the `InternalName`) in addition to its name in the user's language. In the case that an empty string is given for the internal name, it is assumed that the English name (`eName`) is the same as the locale specific name (`aName`).

```
procedure Add(const aName: string; target: TTarget;
              const aHint: string = ''; const eName: string = '';
```

```
virtual;
```

All this requires considerable change to ParserConsts.pas. The target names must be specified twice: once as resource strings that can be translated and once as literal constants that cannot be translated:

```
const
  SC_abs = 'abs';
  SC_acos = 'acos';
  ...
resourcestring
  SR_abs = 'abs';
  SR_acos = 'acos';
    ...
```

Then adding the targets to the list looks like this

```
procedure LoadInternalTargets;
begin
  InternalTargets.Clear;
  InternalTargets.Add(SR_sin, _sin.create(1), S_sin_Hint, SC_sin);
  InternalTargets.Add(SR_cos, _cos.create(1), S_cos_Hint, SC_cos);
  ...
```

while, if `NO_NLS2` is defined, it will look like this:

```
procedure LoadInternalTargets;
begin
  InternalTargets.Add(SC_sin, _sin.create(1), S_sin_Hint);
  InternalTargets.Add(SC_cos, _cos.create(1), S_cos_Hint);
  ...
```

Including the English name in each target, makes it possible to encode the target name, that is translate it to English. To help decode, (translate from English to the user's language) a function, `InternalName`, is added to `TTargetList`:

```
type
  TCompareProc = function(const s1, s2: string): boolean;

function TTargetList.InternalName(const eName: string; out aName:
string): boolean;
var
  i: integer;
  comp: TCompareProc;
begin
  if CaseSensitive then
    comp := @AnsiSameStr
  else
    comp := @AnsiSameText;
  for i := 0 to FList.Count-1 do begin
    result := comp(TTarget(FList.Objects[i]).InternalName, eName);
    if result then begin
      aName := FList[i];
```

```
      exit;
    end;
  end;
  aName := '';
end;
```

The function searches all targets in a list for one with an internal name equal to `eName` and if found returns the target's name in the user language in variable `aName`. Of course the encode and decode functions must be adjusted.

```
function TMathTokenizer.Encode(const source: string): string;
var
  tempTokenizer: TMathTokenizer;
  s: string;
  p: integer;
begin
  result := '';
  tempTokenizer := TMathTokenizer.create;
  try
    tempTokenizer.FormatSettings := FormatSettings;
    tempTokenizer.Source := source;
    while tempTokenizer.NextToken <> ttEOS do begin
      s := tempTokenizer.Token;
      if tempTokenizer.TokenType = ttSeparator then
        s := EnFormatSettings.ListSeparator
      else if (tempTokenizer.TokenType = ttIdentifier) then
        s := tempTokenizer.Target.InternalName
      else if tempTokenizer.TokenType = ttNumber then begin
        p := pos(DecimalSeparator, s);
        if p > 0 then
          s[p] := EnFormatSettings.DecimalSeparator;
      end;
      result := result + s;
    end;
  finally
    tempTokenizer.free;
  end;
end;

function TMathTokenizer.Decode(const source: string): string;
var
  tempTokenizer: TMathTokenizer;
  s, ns: string;
  p: integer;
begin
  result := '';
  tempTokenizer := TMathTokenizer.create;
  try
    tempTokenizer.FormatSettings := enFormatSettings;
    tempTokenizer.Source := source;
    while tempTokenizer.NextToken <> ttEOS do begin
      s := tempTokenizer.Token;
```

```
      if tempTokenizer.TokenType = ttSeparator then
        s := ListSeparator
      else if tempTokenizer.TokenType = ttNumber then begin
        p := pos(enFormatSettings.DecimalSeparator, s);
        if p > 0 then
          s[p] := DecimalSeparator;
      end
      else if tempTokenizer.TokenType = ttUnknownIdentifier then
begin
        if InternalTargets.InternalName(s, ns) then
          s := ns;
      end
      result := result + s;
    end;
  finally
    tempTokenizer.free;
  end;
end;
```

5.3 Duplicate names

It is not possible to add a target to `TTargetList` that has the same name as a previously added target. This is because the `Duplicates` field of the internal `TStringList` holding targets is set to `dupError` which means that an exception will be raised. In the previous version of the parser, the external list goes further and checks any added name against the list of internal targets if `InternalList` is assigned.[4] This approach is no longer feasible because the problem of duplicate names is exacerbated if the names of built-in targets are translated. What will happen should a user with the French locale define a variable named `sqrt`? Recall that the built-in square root function is called `racine` in French so `sqrt` is not a duplicate name. For that user and any other with the French locale it will be possible to save the variable and its value to a file and recover it later. But a user with an English locale would not be able to read that file as a duplicate name error will be signalled when trying to add `sqrt` to the list of external targets.

One partial solution is to check the prospective names of variables against both the translated list of built-in targets and against the list of English target names. Of course that means that there could be over one hundred unacceptable variable names. Furthermore, the solution is far from perfect. Suppose that the built-in function `sqrt` is translated to `trqs` in locale XX. Someone creates, without any problem, a variable `trqs` in a French or English version of a program using the parser and saves it to a file. Then the file becomes unreadable for anyone with locale XX. The only way to avoid this problem is to create all allowed translations of

4 This comment does show a possible error: a target with a name already used in `ExternalTargets` can be added to `InternalTargets` without raising an error. I can't think of a reasonable scenario where this would happen but then I tend to never add targets to `InternalTargets`.

built-in names and to prohibit the use of all these names when creating a variable. This is clearly not desirable and probably not even feasible.

A true solution is to adopt a “magic” character as a prefix of all user defined variables. If the sigil is ‘_’, an assignment statement would look like `def _a := pi/2`, and the variable `_a` could then be used in expression from then on. While not very elegant, this approach works well no matter the number of locales present or added in the future and actually simplifies the implementation as the need for checking for duplicates of built-in functions disappears. But of course, no built-in function must start with the sigil for this to work. In essence, two distinct name spaces are created.

Another possibility is to allow user defined variables with a name equal to a built-in target. Users will have to exercise caution when importing files created by others and check the name of each user defined variable to ensure it is not the same as a built-in target name. This is obviously not a perfect solution, but code that checks the list of imported variables and alerts the user to any redefined built-in targets would help prevent problems. And further help would be provided if use of a sigil were optional. The programmer must not use the sigil as a prefix of any for built-in name. The user should be encouraged to use the prefix for all targets to be saved to a file meant to be shared with others.

The latter possibility is the default approach used here. However by defining the `FORCE_SIGIL` directive, the better solution is also available. There is no longer a need to define the `TExternalTargetList` descendant, `ExternalTargets` will be a `TTargetsList`. The `Add` function of the latter class is straight forward:

```
procedure TTargetList.Add(const aName: string; target: TTarget;
              const aHint: string = ''; const eName: string = '');
begin
  target.Hint := aHint;
  if ename <> '' then
    target.InternalName := eName
  else
    target.InternalName := aName;
  try
    FList.AddObject(aName, target);
  except
    target.Free;
    Raise;
  end;
end;
```

In the distributed code, the private boolean field `FIsInternalList` is added to identify the list of internal targets and assertions are added to the `Add` method just before the try-except lines to ensure that built-in target names do not begin with a sigil and, if `FORCE_SIGIL` is defined, that external target names do begin with a sigil:

```
  {$IFDEF DEBUG}{$IFnDEF NO_SIGIL}
  if FIsInternalList then
    assert(aName[1] <> SSigil, 'Built-in name must not start with
"_"')
  {$IFDEF FORCE_SIGIL}
  else
    assert(aName[1] = SSigil, 'External name must start with "_"')
  {$ENDIF};
  {$ENDIF}{$ENDIF}
```

The verification that an external target's name must begin with the sigil is an assertion that will only be performed in debugging code. Programmers that want to enforce the use of the '_' prefix will have to perform checks before calling a target list's `Add` method. A utility boolean function named `ValidUserName` is provided to perform this task. It verifies that the specified name is an identifier and if `FORCE_SIGIL` is defined it also verifies that it begins with the required prefix.

5.4 Caleçon

While the code constructed so far may be simple, it nevertheless defines a useful parser. The application **Caleçon** uses that parser. In fact, **Caleçon** is actually little more than the demonstration program with the addition of a history of sources that have been evaluated and some cut and paste functionality. Surprisingly, **Caleçon** has proven quite useful for those tasks where many complicated and similar expressions need to be evaluated. I have used it often when marking students homework which included more or less complicated calculations. I wanted to pinpoint where an error occurred when a student used a correct formula but ended up with the wrong answer. I too easily made hard to find mistakes using a (hard or soft) calculator to replicate the student's work and spreadsheets were overkill.

The second half of **Caleçon**, the conversion utility, was inspired by a demonstration program included with recent versions of **Delphi**. I have simplified the interface, perhaps too much? When I first rewrote **Caleçon** in **Free Pascal**, I found errors in the conversion routine. I could not quite follow its logic to be able to fix it and did not want to copy the **Delphi** code which was too C'ish for my taste. I ended up writing my own conversion unit. Given the constant improvement of **Free Pascal**, the conversion unit may now be correct. If not, my own code needs to be cleaned up. When either of those things are done, I will probably release the code for the application. In the meantime, binaries are available.

6 Appendix: List of built in functions, constants and keywords.

This is the list of all built in functions, constants and keywords found in the version 3 and version 4 of the parser.

The second column entitled ‘Description’ is the English language content of the `Hint` property of the target in version 4 of the parser if the conditional directive `TARGET_HINT` is defined. The third column is the type of the target which can be a a keyword (`TKeyword`), a constant (`TConstant`) with the shown value, or a function. The fourth column, only relevant for functions, is the number of parameters or arguments of the function. If the function allows for a variable number of parameters (`TVariableFunction`) then the inequality in the column indicates the minimum number of parameters. If the function has a fixed number of arguments (`TFunction`) then the Params column gives that number.

Identifier	Description	Type	Params
abs	Returns the absolute value	function	1
acos	Returns the inverse cosine	function	1
acosh	Returns the inverse hyperbolic cosine	function	1
acot	Returns the inverse cotangent	function	1
acoth	Returns the inverse hyperbolic cotangent	function	1
acsc	Returns the inverse consecant	function	1
acsch	Returns the inverse hyperbolic cosecant	function	1
asec	Returns the inverse secant	function	1
asech	Returns the inverse hyperbolic secant	function	1
asin	Returns the inverse sine	function	1
asinh	Returns the inverse hyperbolic sine	function	1
atan	Returns the inverse tangent	function	1
atanh	Returns the inverse hyperbolic tangent	function	1
avg	Returns the average of values in the list	function	> 0
ceil	Returns the smallest larger or equal integer	function	1
clamp	Returns the value clamped to bounds	function	3
cos	Returns the cosine	function	1
cosh	Returns the hyperbolic cosine	function	1
cot	Returns the cotangent	function	1
coth	Returns the hyperbolic cotangent	function	1
csc	Returns the cosecant	function	1
csch	Returns the hyperbolic cosecant	function	1
deg	Returns radians as degrees	function	1

Identifier	Description	Type	Params
div	Infix integer division	keyword	
e	Euler's number	constant = 2,7182818285	
exp	Returns the exponential	function	1
fact	Returns the factorial	function	1
floor	Returns the largest smaller or equal integer	function	1
int	Returns the integer part	function	1
ln	Returns natural logarithm	function	1
log	Returns the logarithm with given base	function	2
log10	Returns the common logarithm	function	1
max	Returns the greatest value in the list	function	> 0
min	Returns the smallest value in the list	function	> 0
mod	Infix modulo (division remainder)	keyword	
odd	Returns 1 if odd, 0 if even	function	1
pi	The ratio of the circumference of a circle to its diameter	constant = 3,1415926536	
poly	Returns the value of the polynomial	function	> 1
rad	Returns degrees as radians	function	1
round	Returns the nearest integer	function	1
sec	Returns the secant	function	1
sech	Returns the hyperbolic secant	function	1
sgn	Returns the sign	function	1
sin	Returns the sine	function	1
sinh	Returns the hyperbolic sine	function	1
sqr	Returns the square	function	1
sqrt	Returns the square root	function	1
ssq	Returns the sum of squares of values in the list	function	> 0
tan	Returns the tangent	function	1
tanh	Returns the hyperbolic tangent	function	1
var	Returns the estimated variance	function	> 1
varp	Returns the population variance	function	> 0

7 Bibliography

Aiken, Alex, *Compilers*, https://www.coursera.org/course/compilers

Anez, Juanco (2006) *DUnit: An Xtreme testing framework for Borland Delphi programs*, http://dunit.sourceforge.net/

Boute, Raymond T. (1992) "The Euclidean Definition of the Functions Div and Mod", *ACM Transactions on Programming Languages and Systems* Volume 14 Issue 2, pp 127-144. http://dl.acm.org/citation.cfm?id=128862.

Crenshaw, Jack W. (1988-1995) *Let's Build a Compiler*, Available from many sites on the Web, a nicely formatted version is available at http://www.stack.nl/~marcov/compiler.pdf

Davis, Mark *Unicode Newline Guidelines*, http://unicode.org/reports/tr13/tr13-9.html

Deslierres, Michel (2016a), *Free Pascal/Lazarus gotchas*. On line http://www.sigmdel.ca/michel/program/gotcha_en.html.

Deslierres, Michel (2016b), *Format Settings's List Separator*. On line http://www.sigmdel.ca/michel/program/list_sep_en.html.

Goldberg David (1991), "What Every Computer Scientist Should Know about Floating-Point Arithmetic", *ACM Computing Surveys*, vol. 23, no 1. Can be found at http://perso.ens-lyon.fr/jean-michel.muller/goldberg.pdf

Jensen, Kathleen & Niklaus Wirth (1975) *PASCAL – User Manual and Report*. New-York: Springer-Verlag.

Kernighan, Brian W. and P. J. Plauger (1981) *Software Tools in Pascal.* Reading, Massachusetts: Addison-Wesley Publishing Company.

Lazarus bugtracker (2011) *Bug 0019674: Problems with floating point operations (not raising exceptions)* http://bugs.freepascal.org/view.php?id=19674 (consulted 2016/06/23).

Lazarus bugtracker (2014) *Bug 0026649: Exception in TGtk2WidgetSet.AppInit* http://bugs.freepascal.org/view.php?id=26649 (consulted 2016/06/29)

Lazarus Forum (2011) *Topic: Inconsistency between Windows and Linux compiling?* http://forum.lazarus.freepascal.org/index.php/topic,13460.0.html (consulted 2016/06/22).

Lazarus Forum (2014) *Topic: Cross Win32->x86_64-linux works but Win64->x86_64-linux don't* http://forum.lazarus.freepascal.org/index.php/topic,25637.0.html (consulted 2016/06/29).

Lazarus and Free Pascal wiki (2016) *FPC New Features 3.0 – New iosxlocale unit.* http://wiki.freepascal.org/FPC_New_Features_3.0#New_iosxlocale_unit (consulted 2016/07/07).

Lazarus and Free Pascal wiki (2016) *Multiplatform Programming Guide – Gtk2 and masking FPU exceptions.* http://wiki.freepascal.org/Multiplatform_Programming_Guide#Gtk2_and_masking_FPU_exceptions (consulted 2016/06/23).

Lazarus and Free Pascal wiki (2016) *OS X Programming Tips - Locale settings*. http://wiki.freepascal.org/OS_X_Programming_Tips#Locale_settings_.28date_.26_time_separator.2C_currency_symbol.2C_....29 (consulted 2016/07/07).

Leijen, Daan (2001) "Division and Modulus for Computer Scientists.", http://legacy.cs.uu.nl/daan/pubs.html

Montague, Bruce (2013) *Why you should use a BSD style license for your Open Source Project* http://www.freebsd.org/doc/en/articles/bsdl-gpl/article.html (consulted 2013/02/19)

Wirth, Niklaus (1977), "What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions?", *Communications of the ACM*, Vol 20, No 11 pp 822, 823.