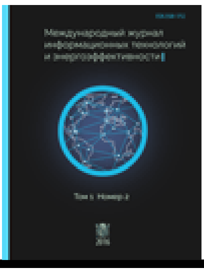
Международный журнал информационных технологий и энергоэффективности

Сайт журнала:
http://www.openaccessscience.ru/index.php/ijcse/



УДК 004.9

РЕАЛИЗАЦИЯ АРИФМЕТИЧЕСКОГО КАЛЬКУЛЯТОРА НА ЯЗЫКЕ OBJECT PASCAL

Коннов Д.В.
Университет Центральной Флориды, Орландо, США (32816, Орландо, Бульвар Центральная Флорида, 4000), e-mail: konnov72@knights.ucf.edu

В статье представлен обзор основных принципов построения арифметического калькулятора, включающего компьютеризированные расчеты по алгоритму маневровой станции и польской нотации PRN. Предложена программная реализация будущего облачного инженерного калькулятора на примере JclExprEval.pas из библиотеки кода JEDI (JCL), одной из наиболее поддерживаемых и актуальных библиотек языка Object Pascal на сегодняшний день. Рассмотрены вопросы токенизации входного выражения, построения дерева разбора, рекурсивного спуска и вычисления дерева. Отмечены требования, необходимые для реализации качественного продукта и приведены примеры реализации кода. Дано обоснование преимуществ использования калькулятора в инженерных расчетах в составе ядра технической системы. На основе этого исследования предложены пути и подходы к разработке будущего многопользовательского системного калькулятора. Обоснована новизна исследования как направление интеграции калькулятора с облачными системами и отделение скриптовых формул приложения от компилируемого кода.

Ключевые слова: Математический эвалюатор, JEDI (JCL), Delphi, токены, дерево синтаксического анализа, дерево синтаксического анализа, синтаксический анализ выражений, рекурсивный спуск, алгоритм сортировочной станции, польская нотация PRN, стек.

IMPLEMENTATION OF ARITHMETIC CALCULATOR IN OBJECT PASCAL

Konnov D.V.
University of Central Florida, Orlando, USA (4000 Central Florida Blvd., Orlando, 32816), e-mail: konnov72@knights.ucf.edu

The article provides an overview of the fundamental principles of constructing an arithmetic calculator including computerized calculations Shunting Yard algorithm and Polish notation PRN. Proposed a software implementation of a future Cloud-based engineering calculator using the JclExprEval.pas as an example from the JEDI code library (JCL), one of the most supported and relevant libraries of the Object Pascal language nowadays. The issues of tokenization of the input expression, building a parse tree, and recursive descent and evaluation of the tree are considered. The requirements necessary for the implementation of a quality product are noted and examples of code implementation are given. The rationale for the advantages of using a calculator in engineering calculations as part of the core of an engineering system is given. Proposed ways and approaches for the development of the future multiuser system calculator based on this research. The novelty of the research is substantiated as a direction of integration of the calculator with Cloud systems and separation of application scripted formulas and compile code.

Keywords: Evaluator, JEDI (JCL), Delphi, tokens, parse tree, Parsing Tree, expression parsing, recursive descent, Shunting Yard Algorithm, Polish notation PRN, stack.

The Problem.

Building an arithmetic calculator involves implementing an algorithm that can calculate mathematical expressions using operations such as addition, subtraction, multiplication, division, and parentheses. Let us immediately define this concept. A calculator is a software device that receives an arithmetic or algebraic expression as a string as input. This apparatus has an interface for adding or determining the values of variables used in the evaluated expression, as well as events that are called during the parsing of the expression, where the values of variables can be substituted dynamically. The result of the Calculator is a floating-point value, i.e., the result of calculating the expression given as input. Let's consider the main stages of developing an arithmetic expression calculator in the Delphi environment using the example of the Evaluator from the JEDI library (JCL) [1], and then we will analyze the detailed aspects of the internal implementation of the parser.

To start, a little history:

The Shunting Yard algorithm [2] is a classic method for parsing infix mathematical expressions and converting them to reverse Polish notation (RPN) [3], also known as postfix notation. RPN is a format in which operators are placed after their operands, making it easier to evaluate expressions using the stack approach.

The algorithm was introduced by Edsger W. Dijkstra [4] in 1961 and is commonly used in compilers, interpreters, and calculator programs. The key idea of the Shunting Yard algorithm is to use two stacks [5], one for operators and one for output (RPN) values, to process an expression respecting operator precedence and associativity.

Here is a step-by-step explanation of how the shunting station algorithm works:

1. While there are tokens to be read:
2. Read a token
3. If it's a number add it to the queue
4. If it's an operator
5. While there's an operator on the top of the stack with greater precedence:
6. Pop operators from the stack onto the output queue
7. Push the current operator onto the stack
8. If it's a left bracket push it onto the stack
9. If it's a right bracket
10. While there's not a left bracket at the top of the stack:
11. Pop operators from the stack onto the output queue.
12. Pop the left bracket from the stack and discard it
13. While there are operators on the stack, pop them into the queue

The resulting RPN expression can be evaluated using the stack approach, in which the operands are placed on the stack, and when an operator occurs, the required number of operands are retrieved from the stack and the result of the operation is placed back on the stack.

The Shunting Yard algorithm ensures that the RPN expression maintains the correct order of operations based on operator precedence and associativity and allows the expression to be evaluated efficiently without using parentheses for grouping.

Let's convert the expression: "(2 + 3) * 4" to RPN

Step 1: Initialize an empty stack for statements and an empty queue (or list) for output (RPN) values (operations are presented in Table 1).

Step 2: Iterate over the tokens of the input infix expression "(2 + 3) * 4":

Table 1 – Step 1 operations

Input Token	**Stack (operators)**	**Output Queue (RPN)**
(	(	
2	(	2
+	+ (	2
3	+ (	2 3
)		2 3 +
*	*	2 3 +
4	*	2 3 + 4

Step 3: Extract all remaining statements from the statement stack and add them to the output queue, the operator (*) remains the last on the stack:

Table 2 – Step 3 operations

Input Token	**Stack (operators)**	**Output Queue (RPN)**
		2 3 + 4 *

The RPN expression – "2 3 + 4 *" is the equivalent of the original infix expression "(2 + 3) * 4" in reverse Polish notation (RPN). In RPN, operators come after their operands, and parentheses are not needed for grouping, because the order of operations is implicitly determined by the position of the operators in the expression.

To evaluate the RPN expression "2 3 + 4 *", a stack-based approach can be used. As follows:

1. Initialize the empty stack.
2. Iterate over an RPN expression from left to right.
3. If the token is a number (operand), then it is placed on the stack.
4. If the token is an operator, then the required number of operands is extracted from the stack, an arithmetic operation is performed, and the result is placed back on the stack.
5. Steps 3 and 4 continue until all tokens have been processed.
6. After processing all the tokens, the result will be at the top of the stack.

Evaluate the RPN expression "2 3 + 4 *" (operations are presented in Table 3):

Table 3 – Operations

Token	**Stack**
2	2
3	2 3
+	5
4	5 4
*	20

After processing all the tokens, the result in the stack is 20. Therefore, the actual result of the expression "2 3 + 4 *" is 20.

Materials and research methods

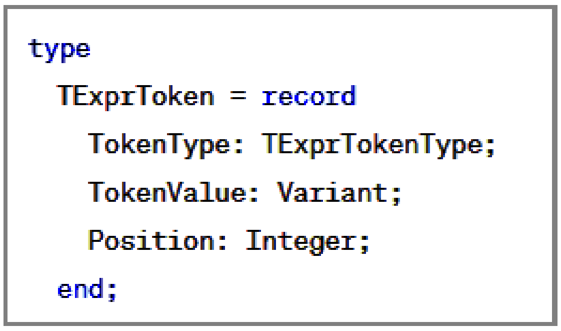
The Object Pascal language in modern days has proven itself as a reliable, high-level programming language that allows you to build systems on the scale of large corporations and enterprises. The development of an arithmetic engineering calculator [6] in this language is a completely natural and essential task of any corporate software product in the Delphi environment [7].

The JclExprEval.pas module in the JEDI Code Library (JCL) provides a flexible *expression evaluator* capable of analyzing and evaluating mathematical expressions. It allows you to dynamically evaluate arithmetic, Boolean, and comparison expressions at runtime. Let's take a closer look at how the JclExprEval.pas parser works. The main functionality of JclExprEval is to parse the input expression, building a *parsing tree* (Parse Tree) [9] and its evaluation. Below are abstracted implementation details that may not be properly aligned with the latest library code but with the purpose of explaining how it all works.

First, the input expression must be tokenized.

Tokenization: This is the division of an input mathematical expression into individual tokens, including numbers, operators, and parentheses [10]. The expression is first tokenized by breaking it down into individual components, such as numbers, operators, functions, and variables. Tokens are the building blocks that the analyzer uses to understand the expression when building a *parse tree*. Each token is represented by a **TExprToken** record that contains information about its type, value, and position in the expression.

The TExprToken record is a fundamental data structure used in the JclExprEval.pas module to represent individual tokens in the parsed expression. It contains information about the type, value, and position of the token in the original expression. The definition of the TExprToken record is shown in Figure 1.



```
type
  TExprToken = record
    TokenType: TExprTokenType;
    TokenValue: Variant;
    Position: Integer;
  end;
```

Figure 1 – Structure of the expression token.

1. TokenType: The TokenType field specifies the type of token, which can be one of the following values (defined in the JclExprEval.pas module):

- etUnknown: Unknown type.
- etNumber: Numeric value.
- etVariable: Variable name.
- etOperator: operator.
- etLeftParenthesis: left parenthesis "(".
- etRightParenthesis: right parenthesis ")".
- etFunction: A token that represents the name of the function.

2. TokenValue: The TokenValue field contains the actual value of the token. Its data type is Variant, which means that it can store different types of values depending on the type of token:

- etNumber: numeric value in the form of Double.
- etVariable: variable name as a string.
- etOperator: operator as a string.
- etFunction: function name as a string.

3. Position: The Position field specifies the position of the token in the original expression as an index. The index is zero-based, which means that the first character in the expression has position 0.

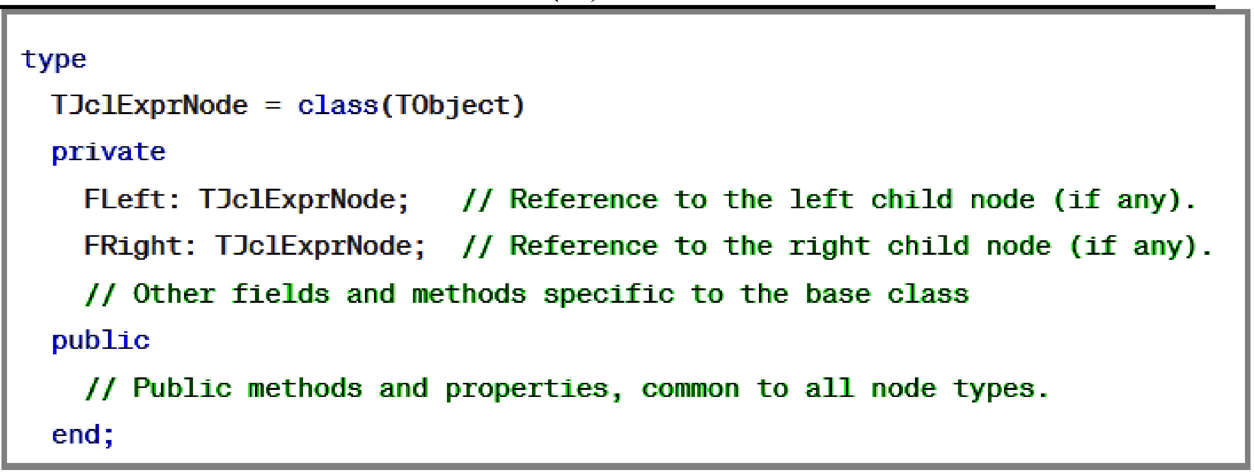
Each token is part of an expression, including numbers, operators, and parentheses, and stores the corresponding properties in a TExprToken record. TExprToken entries are temporarily stored in an internal array in the order in which they appear in the expression. This array of TExprToken entries is used to represent the tokenized expression during the parsing step.

For example, the input expression "2 + 3 * 4" is tokenized, and TExprToken entries are created for each token. The resulting array of TExprToken records might look like this:

[Token(2), Token(+), Token(3), Token(*), Token(4)]

Each element of this array is a TExprToken entry. This is important for the parsing and evaluation process because the Token helps the JclExprEval module understand the structure of the expression and perform the necessary operations to evaluate it. Next, the analyzer starts parsing the token array. When the algorithm encounters each token, it uses the information stored in the token to build the structure of the parse tree and make decisions about how to handle subsequent tokens. Depending on the type of token, the algorithm creates operator nodes, operand nodes, or functional nodes. In the process of processing tokens, the algorithm builds **TJclExprNode** objects and collects these nodes (nodes) into a *parsing tree* (Figure 4).

The structure of TJclExprNode is shown in Figure 2. In JclExprEval.pas, each node *in the parse* tree is represented by a specific class named TJclExprNode. The TJclExprNode class is the base class for various types of nodes in the parser tree, including operator nodes, operand nodes, and functional nodes. TJclExprNode (the base class for all nodes) represents a generic node in the parse tree. Contains properties and methods that are common to all types of nodes. The TJclExprNode class and its subclasses in the parse tree support referencing child nodes through their FLeft and FRight properties. The parse tree is built using these references, which allows nodes to be linked to each other hierarchically. The TJclExprNode class has properties that define the structure of the parse tree and how the nodes are connected:



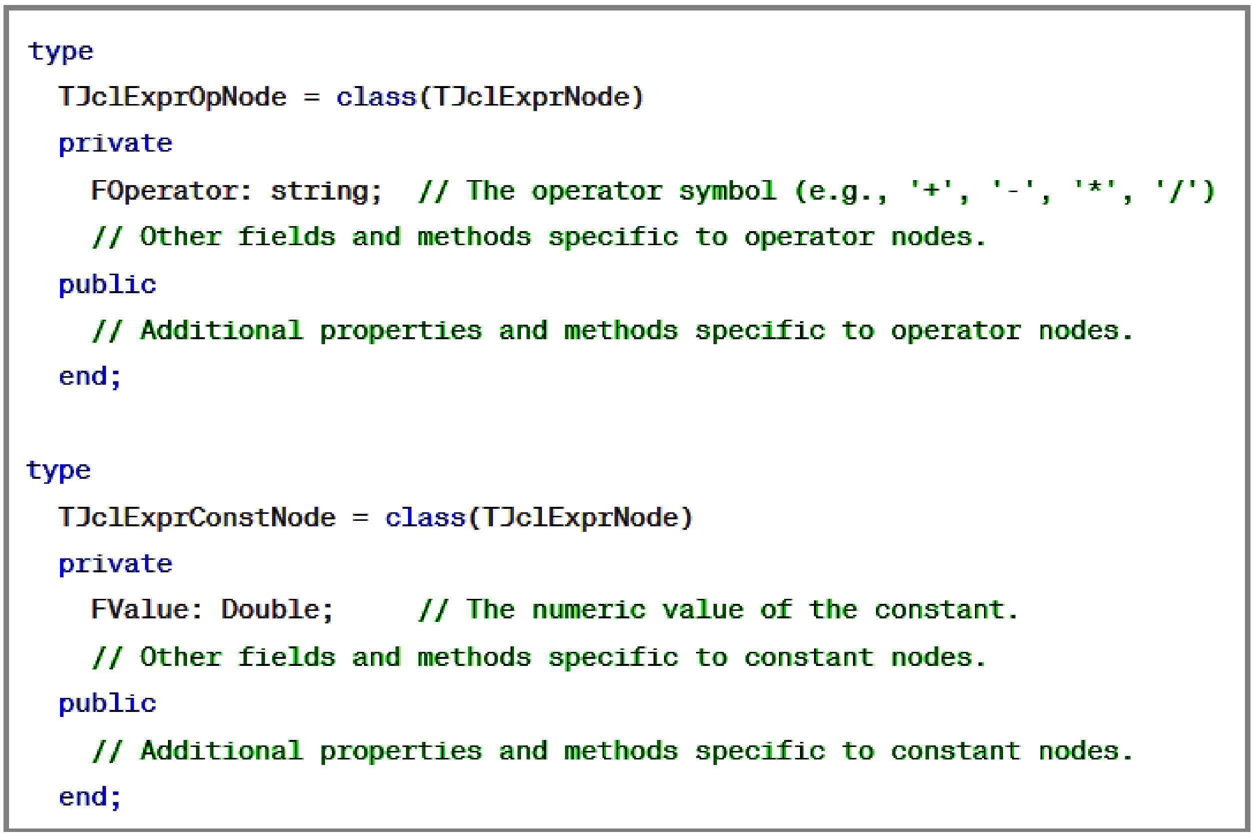
```
type
  TJclExprNode = class(TObject)
  private
    FLeft: TJclExprNode;   // Reference to the left child node (if any).
    FRight: TJclExprNode;  // Reference to the right child node (if any).
    // Other fields and methods specific to the base class
  public
    // Public methods and properties, common to all node types.
  end;
```

Figure 2 – The structure of the node of the parsing tree.

FLeft: This property contains a reference to the left child node of the current node. It allows for a hierarchical organization of nodes, where the left child node is usually the operand, argument, or first operand for the binary operator.

FRight: This property contains a reference to the right child node of the current node. It is used for binary operators, where the right child represents the second operand.

In subclasses of TJclExprNode (Figure 3), Certain types of nodes can extend these properties as needed to meet their specific requirements.



```
type
  TJclExprOpNode = class(TJclExprNode)
  private
    FOperator: string;  // The operator symbol (e.g., '+', '-', '*', '/')
    // Other fields and methods specific to operator nodes.
  public
    // Additional properties and methods specific to operator nodes.
  end;

type
  TJclExprConstNode = class(TJclExprNode)
  private
    FValue: Double;     // The numeric value of the constant.
    // Other fields and methods specific to constant nodes.
  public
    // Additional properties and methods specific to constant nodes.
  end;
```

Figure 3 – An example of declaring classes derived from TJclExprNode nodes.

JclExprEval.pas implements the TJclExprOpNode node class and its descendants, each of which performs a specific arithmetic function, for example:

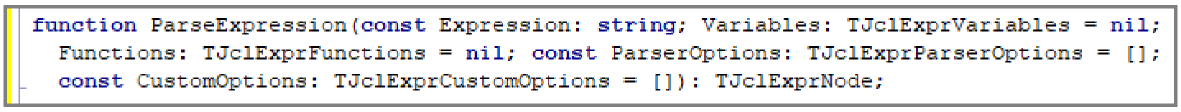
- assignment statement "="
- Boolean operators (e.g., AND, OR)
- relational operators (e.g., >, <, =)
- побитовые операторы (например, AND, OR, XOR)
- shift operators (e.g., SHL, SHR)
- addition operator "+"
- subtraction operator "-"
- Multiplication operator "*"
- division operator "/"
- operator obtaining the remainder by division "%" (mod)
- unary operators (e.g., unary minus)
- Degree Operator "^"
- factorial operator "!"

TJclExprNode also has the following subclasses:

- TJclExprVarNode: variable.
- TJclExprConstNode: constant (numeric value).
- TJclExprFuncNode: represents a function call. Contains properties for storing the function name and argument node references.

Parsing Tree

The next step in processing a tokenized expression is to build a parsing tree (Figure 4). This is the process of constructing a hierarchical representation of the structure of an expression. Expression nodes, put together using **FLeft, Fright** pointers, form a *parsing tree*. A tree is a tree-like data structure in which each node represents an operator or function, and its child elements represent operands or arguments. In the JclExprEval.pas module of the JEDI Code Library (JCL), the recursive descent algorithm **does not use the stack to build the parse tree.** The algorithm directly builds the parse tree, recursively [11] evaluating the expression based on the rules of grammar and the precedence of operators. This is implemented using a recursive function called ParseExpression. This function is responsible for the recursive evaluation of the input expression and the construction of the parse tree. The following is the declaration of the ParseExpression function.



```
function ParseExpression(const Expression: string; Variables: TJclExprVariables = nil;
  Functions: TJclExprFunctions = nil; const ParserOptions: TJclExprParserOptions = [];
  const CustomOptions: TJclExprCustomOptions = []): TJclExprNode;
```

Figure 4 – ParseExpression function declaration.

Function parameters:

- Expression: Input expression for analysis and evaluation.
- Variables: A set of variables that can appear in an expression (optional).
- Functions: A set of user-defined functions that can be used in an expression (optional).
- ParserOptions: additional parameters that control parsing behavior.

- CustomOptions: More configurable options to enhance parsing capabilities.

Return value:

- TJclExprNode: The root *of the parse tree* that represents the input expression.

The ParseExpression function is the entry point for starting the recursive descent process for parsing and building the parse tree. It performs recursive parsing and evaluation of the expression, creating the appropriate nodes and linking them together to form a hierarchical parse tree structure. As soon as the ParseExpression function is called with an input expression, it returns the root of the parse tree, which can later be used for evaluation or other operations on the expression.

For example, with the expression "2 + 3 * 4", the parse tree might look like this:

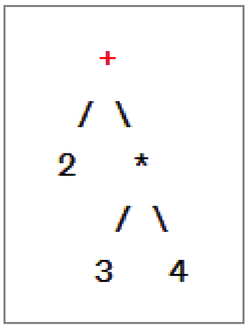


Figure 5 – Parsing tree.

In this example, the + operator node has 2 as the left child node and the * operator node as the right child node. The * operator node, in turn, has 3 as the left child node and 4 as the right child node.

It is easy to see: The nodes of the OPERANDS do not point to any other nodes in the parsing tree, and the nodes of the OPERATORS and FUNCTIONS point to the nodes of the OPERANDS.

Let's summarize the algorithm for constructing a parsing tree:

1. Tokenization: The input expression is tokenized, and each token (e.g., numbers, operators, functions, variables) is represented by a TExprToken record.

2. Building a Parse Tree: Once tokenization is complete, the Recursive Descent algorithm begins to build the parsing tree. It recursively processes tokens, evaluating the expression based on grammar rules and operator precedence.
3. Node creation and linking: When the recursive descent algorithm encounters each TExprToken, it creates the corresponding nodes and links them together as children and parents to form the parse tree structure. The algorithm sets the FLeft and FRight properties of operator nodes based on their operands to properly link them together.
4. Operator precedence: The parsing algorithm considers the priority of operators to ensure that expressions are evaluated according to the correct order of operations (for example, multiplication before addition).

5. Associativity: For operators with the same priority, the parsing algorithm considers their associativity (left-to-right or right-to-left) to maintain the correct order of calculation.
6. Hierarchical relationships: The recursive descent process provides a hierarchical organization of nodes in the parse tree based on the structure of the expression.
7. Root node: After the recursive descent process is complete, the root node of the parse tree represents the entire expression, and the parse tree is ready for evaluation.

Evaluation. Building a parse tree is an intermediate step in the overall process. To get the final result of the expression, the algorithm needs to evaluate the constructed *parsing* tree. The evaluation process involves traversing the tree and applying mathematical operations defined by operators and functions. The algorithm will recursively evaluate subexpressions and combine their results based on the operators and functions encountered during the traversal.

For example, consider the expression "2 + 3 * 4" (Figure 5):

To evaluate the expression, the algorithm will run at the root of the parse tree (the + operator). It will recursively traverse the tree, evaluating the left and right subtrees. The algorithm will first evaluate the operator node * (3 * 4 = 12) and replace the operator node * with its result (12). The updated tree is shown in Figure 6.

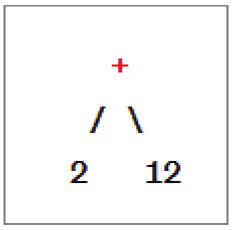
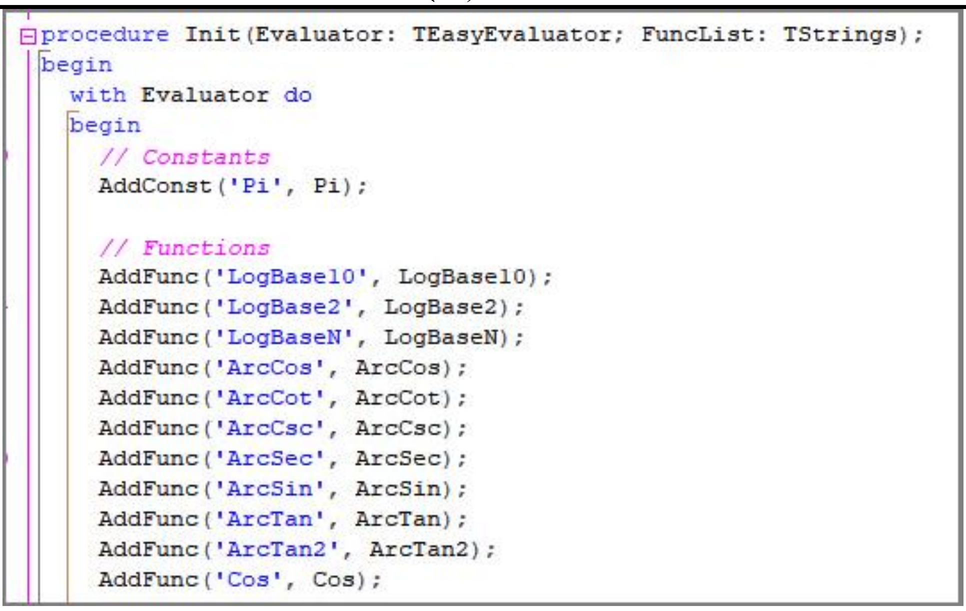


Figure 6 – The Parsing Tree in the process of parsing.

The algorithm will now evaluate the operator node + (2 + 12 = 14) and replace the operator node + with its result (14).

The result of the expression is 14.

It should be noted that to ensure the quality of the calculator, the correct handling of errors of invalid expressions, division by zero, inappropriate parentheses, etc., must be implemented, and support for additional functions such as trigonometric functions, exponentiation, variables, and much more can be added (Figure 7).

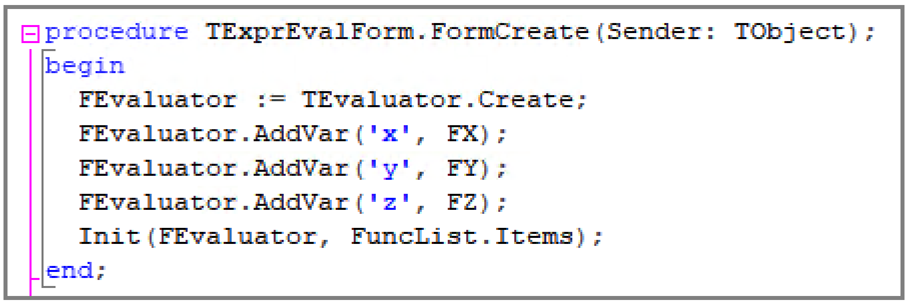


```
procedure Init(Evaluator: TEasyEvaluator; FuncList: TStrings);
begin
  with Evaluator do
  begin
    // Constants
    AddConst('Pi', Pi);

    // Functions
    AddFunc('LogBase10', LogBase10);
    AddFunc('LogBase2', LogBase2);
    AddFunc('LogBaseN', LogBaseN);
    AddFunc('ArcCos', ArcCos);
    AddFunc('ArcCot', ArcCot);
    AddFunc('ArcCsc', ArcCsc);
    AddFunc('ArcSec', ArcSec);
    AddFunc('ArcSin', ArcSin);
    AddFunc('ArcTan', ArcTan);
    AddFunc('ArcTan2', ArcTan2);
    AddFunc('Cos', Cos);
```

Figure 7 – Registration of user-defined functions.

Also, the evaluator implements the ability to register variables (Fig. 8) and provides an interface for their initialization.



```
procedure TExprEvalForm.FormCreate(Sender: TObject);
begin
  FEvaluator := TEvaluator.Create;
  FEvaluator.AddVar('x', FX);
  FEvaluator.AddVar('y', FY);
  FEvaluator.AddVar('z', FZ);
  Init(FEvaluator, FuncList.Items);
end;
```

Figure 8 – Registration of Evaluation variables.

Implementing the User Interface

An illustrative example of the user interface (Figure 9.), which allows you to enter arithmetic expressions, and evaluate and display the results can be found in the examples\common\expreval folder of the JEDI library (JCL). An example of using the Calculator is shown in Figure 10.

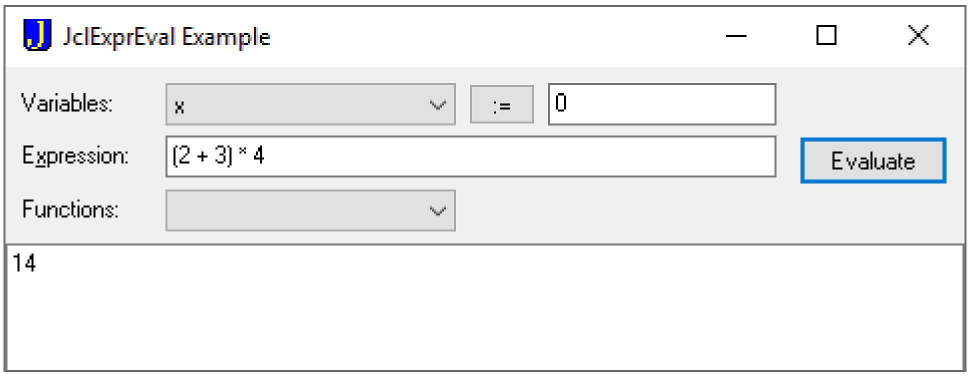
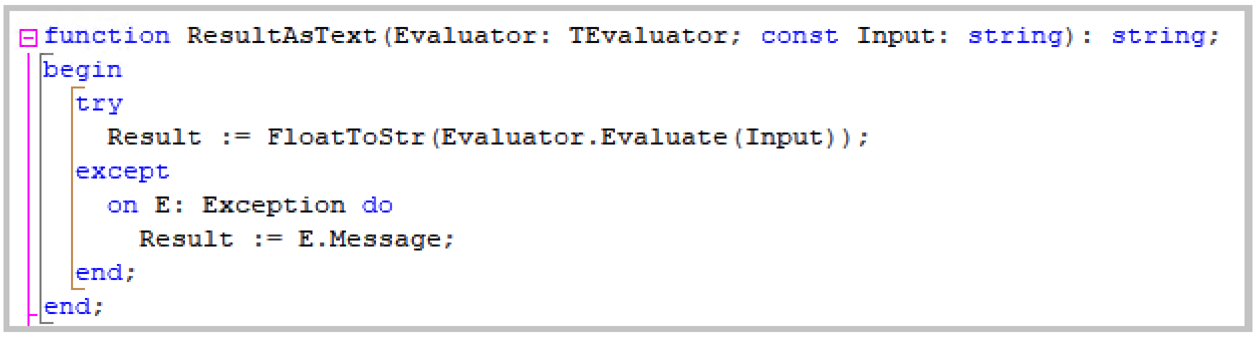


Figure 9 – Graphical user interface of the Calculator.



```
function ResultAsText(Evaluator: TEvaluator; const Input: string): string;
begin
  try
    Result := FloatToStr(Evaluator.Evaluate(Input));
  except
    on E: Exception do
      Result := E.Message;
  end;
end;
```

Figure 10 – An example of using the Evaluator.

Discussion of the results

One of the main requirements for engineering computing systems these days is, of course - performance. It is worth noting that the performance of this calculator on the local Desktop system is quite satisfactory and does not raise questions from the user. However, the implementation of such a calculator as part of a multi-user system can cause performance problems [12]. Therefore, the developer will have to think about the implementation of the system parallelism [13] and base further development on its principles.

To improve calculation speed, we can consider applying the following approaches:

1. **Reduce Expression Complexity**: Complex expressions with many nested functions and operators can lead to slower evaluation. We want to simplify expressions where possible to reduce the number of operations.
2. **Precompile Expressions**: If we have expressions that are evaluated frequently, consider precompiling them into a reusable format. This can save time on parsing and tokenizing the expression each time it's evaluated.
3. **Caching**: If our calculations involve the same set of variables but different expressions, we may consider caching the values of variables that don't change often. This can reduce the overhead of variable lookup.
4. **Reuse Instances**: If we need to evaluate multiple expressions in the same context, we must reuse the same instance of the expression evaluator rather than creating a new one for each evaluation. Creating and initializing instances can be time-consuming.
5. **Use Inline Variables**: If possible, assign values to variables before evaluating the expression. This can help simplify the expression and reduce the number of times variables are looked up.
6. **Optimize Your Expressions**: Depending on the specific expressions we're evaluating, there might be opportunities for optimization. For example, replacing expensive operations with equivalent but faster operations.
7. **Use Compiled Code**: Some expression evaluation libraries allow us to compile expressions into native machine code for faster execution. The JclExprEval or future developing calculator based on such principles should support expression compilation.
8. **Evaluate in Bulk**: If we need to evaluate the same expression for multiple inputs, consider batching the evaluations together. This can take advantage of any optimizations that the library has for processing multiple inputs at once.

Considering the items above, the author of this article forked a new branch of the project called **unCalc (Universal Calculator)** and the project is under development. https://github.com/dima72/unCalc [14]

Conclusion. The advantages of using an arithmetic calculator as part of engineering calculation systems are more than obvious. Firstly, a dynamically interpreted mathematical expression can always be stored as a string, whether it is just a file or a database, as part of any program, and therefore be available for editing by the user, which in turn greatly facilitates the support and development of any software product. Now you do not need to recompile the entire project to fix an error in the formula. Secondly, the implementation of the apparatus of the mathematical calculator is compact, the reduction of the manual code for processing calculations, formulas always improving the quality of the product, allows you to separate the apparatus of calculations from the formulas themselves. Thirdly, knowledge of the construction of an arithmetic calculator helps to achieve a deeper understanding of the development of interpreters, compilers, and universal software. Considering current development trends in information technology, the language of the calculator implementation is of secondary importance. This calculator can be implemented as part of frameworks and languages that integrate with the cross-platform .NET framework, such as Oxygene and Hydra from RemObjects [15] or on the TMS XData platform [16]. The novelty of this approach lies in the provision of distributed engineering computing on the Cloud.

Список литературы

1. Библиотека кода JEDI распространяется на условиях публичной лицензии Mozilla (MPL). // https://github.com/project-jedi/jcl
2. Конструкция компилятора Wikipedians // PediaPress 189с.
3. Джон Левин flex & bison: Инструменты для обработки текста // O'Reilly Media 2009, 84с.
4. Оле-Йохан Даль, Эдсгер В. Дейкстра, Чарльз А. Р. Хоар Структурированное программирование // Acad. Пресса 1980
5. Джим Кеог, Дэвидсон Структуры данных: принципы и фундаментальные основы // Dreamtech Press 2004
6. Банников Н.А. Как писать переводчики. // http://www.stikriz.narod.ru/art/Interp.htm (дата посещения: 08/02/2023)
7. Ксавье Пачеко, Стив Тейшейра Руководство разработчика Borland Delphi 6 // Sams 2001
8. Марко Кэнти осваивает Borland Delphi // Wiley 2005, 913с.
9. Паллави Виджай Чаван, Ашиш Джадхав Теория автоматов и формальные языки // Elsevier Science 2023 112с.
10. Ханнес Хапке, Коул Ховард, Хобсон Лейн Обработка естественного языка в действии // Мэннинг 2019
11. Мануэль Рубио-Санчес Введение в рекурсивный Pr.
12. Björn Andrist, Viktor Sehr C++ High Performance Boost and Optimize the Performance of Your C++17 Code // Packt Publishing, 2018
13. Dan C. Marinescu Cloud Computing: Theory and Practice // Elsevier Science 2022
14. Dmitry Konnov unCalc // https://github.com/dima72/unCalc (date of visit: 08/14/2023)
15. RemObjects Software // https://www.remobjects.com (date of visit: 08/02/2023)

16. Delphi framework for multi-tier REST/JSON HTTP/HTTPS application server development and ORM remoting. // https://www.tmssoftware.com/site/xdata.asp (date of visit: 08/02/2023)

References

1. The JEDI Code Library is distributed under the terms of the Mozilla Public License (MPL). // https://github.com/project-jedi/jcl
2. By Wikipedians Compiler Construction // PediaPress p.189
3. John Levine flex & bison: Text Processing Tools // O'Reilly Media 2009, p. 84
4. Ole-Johan Dahl, Edsger W. Dijkstra, Charles A. R. Hoare Structured programming // Acad. Press 1980
5. Jim Keogh, Davidson Data Structures: Principles and Fundamentals // Dreamtech Press 2004
6. Bannikov N.A. How to write interpreters. // http://www.stikriz.narod.ru/art/Interp.htm (date of visit: 08/02/2023)
7. Xavier Pacheco, Steve Teixeira Borland Delphi 6 Developer's Guide // Sams 2001
8. Marco Canty Mastering Borland Delphi // Wiley 2005 p.913
9. Pallavi Vijay Chavan, Ashish Jadhav Automata Theory and Formal Languages // Elsevier Science 2023 p.112
10. Hannes Hapke, Cole Howard, Hobson Lane Natural Language Processing in Action // Manning 2019
11. Manuel Rubio-Sanchez Introduction to Recursive Programming // CRC Press 2017
12. Björn Andrist, Viktor Sehr C++ High Performance Boost and Optimize the Performance of Your C++17 Code // Packt Publishing, 2018
13. Dan C. Marinescu Cloud Computing: Theory and Practice // Elsevier Science 2022
14. Dmitry Konnov unCalc // https://github.com/dima72/unCalc (date of visit: 08/14/2023)
15. RemObjects Software // https://www.remobjects.com (date of visit: 08/02/2023)
16. Delphi framework for multi-tier REST/JSON HTTP/HTTPS application server development and ORM remoting. // https://www.tmssoftware.com/site/xdata.asp (date of visit: 08/02/2023)