Articles » General Programming » Algorithms & Recipes » Parsers and Interpreters

Evaluation Engine

DonSn, 1 Jun 2008

The Evaluation Engine is a parser and interpreter that can be used to build a Business Rules Engine. It allows for mathematical and boolean expressions, operand functions, variables, variable assignment, comments, and short-circuit evaluation. A syntax editor is also included.

Download Source and Test Applications - 1.04 MB

Evaluation Engine Overview

I am in the process of creating a Rules Engine. I have architected the Rules Engine to allow the rules to be "pre-compiled" or "dynamic". The "pre-compiled" rules are simply rules that are programmed in a .Net language and invoked by the Rules Engine using reflection. The "dynamic" rules are interpreted at execution time by the Rules Engine. The "dynamic" rules offer great flexibility because they can be changed very easily.

This article is not about the Rules Engine; it is about the parser/interpreter that is built into the Rules Engine that evaluates the dynamic rules. This piece of the code that evaluates the dynamic rules is called the Evaluation Engine. Included in the download is the Evaluation Engine source code and test application. The main test application (called the Rules Calculator) allows you to enter in the rule syntax, parse the syntax, and evaluation the results.

Evaluation Engine

Before going into too much detail on how the Evaluation Engine works, I thought it would be better to show you what it does. After you see what the Evaluation Engine can do, I will explain how it works. As I mentioned previously, the test application that is included is called the Rules Calculator. The Rules Calculator is simply a "wrapper" around the Evaluation Engine. The Rules Calculator allows you to type in Rules Syntax (for lack of a better name, I'll call the syntax "Rules Basic"). Next, the Rules Calculator passes the Rules Basic syntax to the Evaluation Engine for evaluation.

Parsing and Tokens

The job of the Evaluation Engine is to take the Rules Basic string (or .Rule file), evaluate the contents and perform the action. Most compilers and interpreters go through a process called parsing. The parsing process needs to look at the Rules Basic string and pull out the important pieces of information. The important pieces of information the parser identifies are called "tokens". Each token gets identified as a certain type of token (or classification). Each token is classified as one of the following:

Token_Operand	An operand is a value or a variable in the Rules Basic syntax. For example, in the mathematic equation "1 + 2 = 3", the items 1, 2, and 3 are operands. Similarly, in the equation "a + -b >= c" the operands are a, b, and c. Obviously, there is a difference between the operands 1, 2, 3 and a, b, c. That is, the first set of operands are Integer values and the second set are variables. The point is that the operand tokens can be further classified by types. The Evaluation Engine classifies the Operand Tokens as one of the following: • Token_DataType_Variable: The operand token is a variable. • Token_DataType_Int: The operand token is an integer/whole number.

	• Token_DataType_Date: The operand token is a date. • Token_DataType_Double: The operand token is numeric/double. • Token_DataType_String: The operand token is text. • Token_DataType_Boolean: The operand token is true or false. • Token_DataType_NULL: The operand token is null.
Token_Operand_Function_Start	The Rules Basic syntax allows for function on operands. This token indicates that the parser has found the start of an operand function. All operand functions in the Rules Basic syntax utilize square brackets. For example, Avg[], Year[], Join[] are 3 examples of operand functions that are available in Rules Basic. Operand functions can take operands are parameters. The Evaluation Engine does some data type checking. For example, Year[5.20.1999] operand function required an operand token of type Token_DataType_Date (notice that the date delimiter is "."). The same operand function with the wrong data type operand will throw an error, i.e., Year["Hello"] will cause an error.
Token_Open_Parenthesis	The Evaluation Engine follows the mathematical order of operations. This order of operations can be changed by using parenthesis. This token indicates that the Evaluation Engine found an open parenthesis: (
Token_Close_Parenthesis	The Evaluation Engine has found a close parenthesis.
Token_Operand_Function_Stop	The Rules Basic syntax allows for operand function. The parameters for operand functions are identified in square brackets. This token indicates that the Evaluation Engine found the end of an Operand Function.
Token_Operand_Function_Delimiter	Operand functions can take multiple parameters. The parameters in an operand functions are separated by commas. This token indicates that the Evaluation Engine has found a comma in a Operand Function
Token_Operator	Operators are applied to operands. For example, when the "+" operator is applied to the operands 1 and 1, the result is 2. This token indicates that the Evaluation Engine has found an operator.
Token_Assignemt_Start	The Rules Basic syntax allows for assignment of variables. Variables can be assigned static values (such as "Hello") or they can be assigned Rules Basic syntax (such as "1 + 1"). The assigned variables can also participate in other Rules Basic syntax. The assignment operator is := Examples of assignments are seen in the examples below. Therefore, when the Evaluation Engine encounters a := in the Rules Basic string, a Token_Assignemt_Start gets created in the collection of tokens.
Token_Assignment_Stop	The semicolon (;) indicates the end of an assignment in Rules Basic syntax. When the Evaluation Engine detects ; a Token_Assignment_Stop token gets created.

One more important piece of information: When the Evaluation Engine parses the Rules Basic string, every token gets assigned a token type and a token data type. For example, the token "Hello" will get assigned the Token Type = Token_Operand and the Token Data Type = Token_DataType_String. If you forget to the double quotes around the string "Hello", the parser will classify Hello as a Token_Operand and data type Token_DataType_Variable. Since every token gets classified with a token type and token data type, how does the parser classify the data type of a = token, or a + token, or a (token? The token + gets classified as token type Token_Operator and data type Token_DataType_None. I did not mention the "None" data type in the tables above because I wanted to provide this example first. Therefore, the operand tokens can be classifies as Integer, Date, Double, String, Boolean, Null, Variable or None.

Let's work through an example of parsing and tokens. We will keep the Rules Basic syntax simple: (1 + 3) * Avg[3,4,2+3]

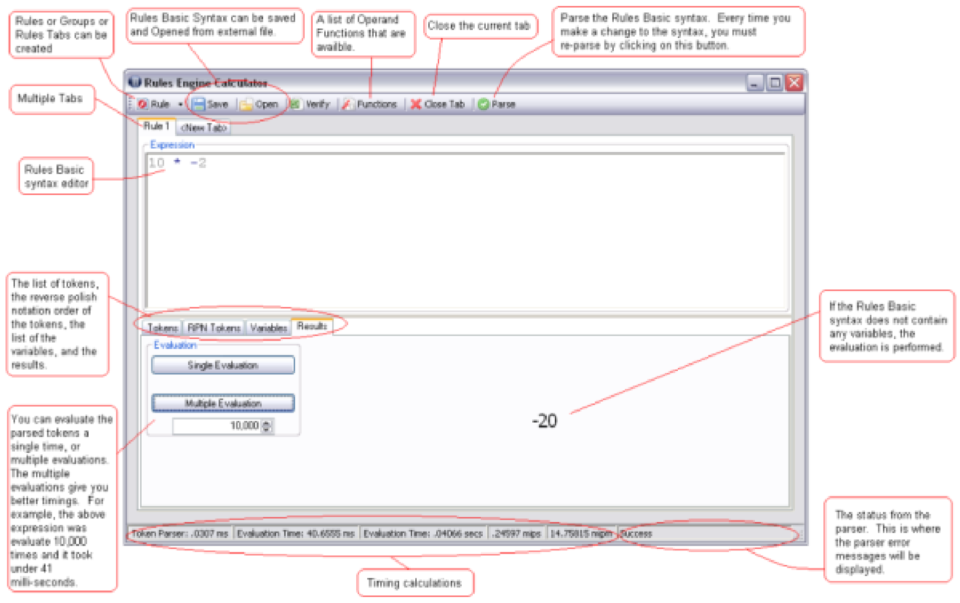
Here are the tokens that get identified in the above example:

Token	Token Type	Token Data Type
(	Token_Open_Parenthesis	Token_DataType_None
1	Token_Operand	Token_DataType_Int
+	Token_Operator	Token_DataType_None
3	Token_Operand	Token_DataType_Int
)	Token_Close_Parenthesis	Token_DataType_None
*	Token_Operator	Token_DataType_None
Avg[	Token_Operand_Function_Start	Token_DataType_None
3	Token_Operand	Token_DataType_Int
,	Token_Operand_Function_Delimiter	Token_DataType_None
4	Token_Operand	Token_DataType_Int
,	Token_Operand_Function_Delimiter	Token_DataType_None
2	Token_Operand	Token_DataType_Int
+	Token_Operator	Token_DataType_None
3	Token_Operand	Token_DataType_Int
]	Token_Operand_Function_Stop	Token_DataType_None

After the Evaluation Engine identifies and classifies all the tokens in the Rules Basic syntax, the tokens are arranged in a certain order (more on this later) to make the evaluation easier.

Examples

Time for some examples. I'll start with simple examples and then introduce some more complex examples:



Rules or Groups or Rules Tabs can be created

Rules Basic Syntax can be saved and Opened from external file.

A list of Operand Functions that are availble.

Close the current tab

Parse the Rules Basic syntax. Every time you make a change to the syntax, you must re-parse by clicking on this button.

Multiple Tabs

Rules Basic syntax editor

The list of tokens, the reverse polish notation order of the tokens, the list of the variables, and the results.

If the Rules Basic syntax does not contain any variables, the evaluation is performed.

You can evaluate the parsed tokens a single time, or multiple evaluations. The multiple evaluations give you better timings. For example, the above expression was evaluate 10,000 times and it took under 41 milli-seconds.

```
Token Parser: .0307 ms | Evaluation Time: 40.6555 ms | Evaluation Time: .04066 secs | .24597 mips | 14.75815 mipm | Success
```

The status from the parser. This is where the parser error messages will be displayed.

Timing calculations

This is a simple example that shows you the Rules Calculator test application. The Rules Calculator allows you to create multiple tabs to run multiple expressions. I did this so that you can break down a large complicated expression into smaller expressions and evaluate the smaller expressions in their own tabs.

The Rules Basic syntax allows you to perform mathematical calculations. It allows you to do order of operations. The valid mathematical operators are:

<t />

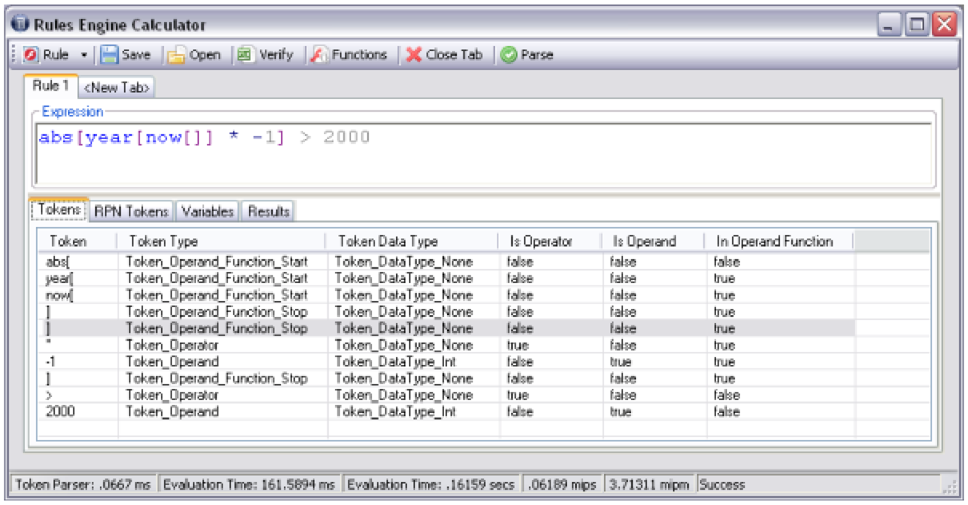
^	Exponents: 5 ^ 2 = 25
*	Multiplication: 5 * 5 = 25
/	Division: 25 / 5 = 5
%	Modulus/Remainder: 10 % 3 = 1
+	Addition: 5 + 5 = 10
-	Subtraction: 10 - 5 = 5. The - symbol is also used to represent negative numbers. For example 1+-1=0

Of course, the Rules Basic syntax also allows for logical and Boolean operators:

<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
=	Equal to
<>	Not equal to

and	Logical and
or	Logical or

The Evaluation Engine allows you to perform mathematical calculations, string manipulations, date calculations, and Boolean logical. Operand functions can be embedded within Operand functions (within operand functions....). Mathematical expressions (with order of operation) can be embedded within operand functions. In the example below, notice that mathematical expression is embedded within operand functions which are embedded in other operand function and compared against the integer value 2000:

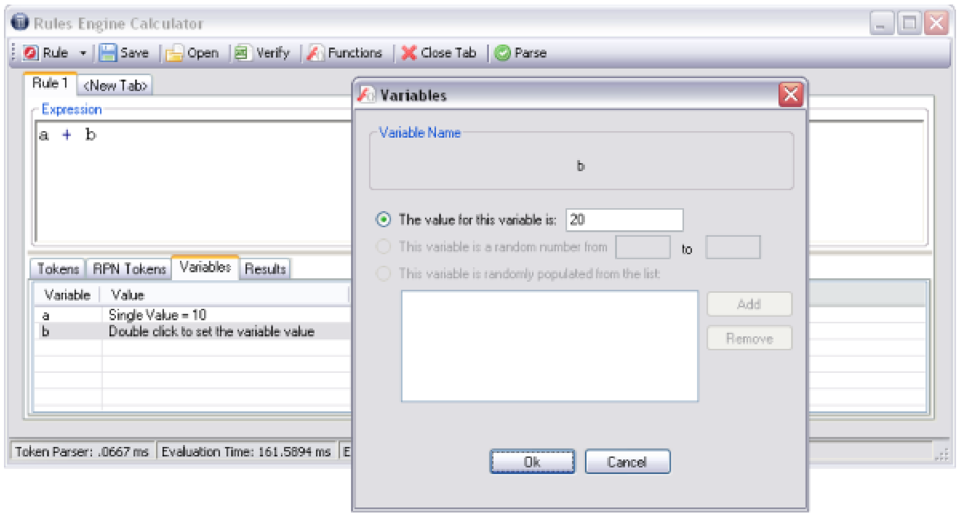


```
abs[year[now[]] * -1] > 2000
```

Token	Token Type	Token Data Type	Is Operator	Is Operand	In Operand Function
abs[	Token_Operand_Function_Start	Token_DataType_None	false	false	false
year[	Token_Operand_Function_Start	Token_DataType_None	false	false	true
now[	Token_Operand_Function_Start	Token_DataType_None	false	false	true
]	Token_Operand_Function_Stop	Token_DataType_None	false	false	true
]	Token_Operand_Function_Stop	Token_DataType_None	false	false	true
*	Token_Operator	Token_DataType_None	true	false	true
-1	Token_Operand	Token_DataType_Int	false	true	true
]	Token_Operand_Function_Stop	Token_DataType_None	false	false	true
>	Token_Operator	Token_DataType_None	true	false	false
2000	Token_Operand	Token_DataType_Int	false	true	false

Token Parser: .0667 ms | Evaluation Time: 161.5894 ms | Evaluation Time: .16159 secs | .06189 mips | 3.71311 mipm | Success

Here's another simple example using variables. After you parse the Rules Basic syntax, the Rules Calculator realizes that the expression contains variables, therefore, it automatically brings you to the variables table. You can set the value of a variable by double clicking on the variable in the list view and setting its value. After all the variables are set, you can click on the results tab and execute a "Single Evaluation". Note that you can change the values of the variables at any time and re-evaluate the result. However, anytime you change the syntax, you need to click on the "Parse" button in the main toolbar:



```
a + b
```

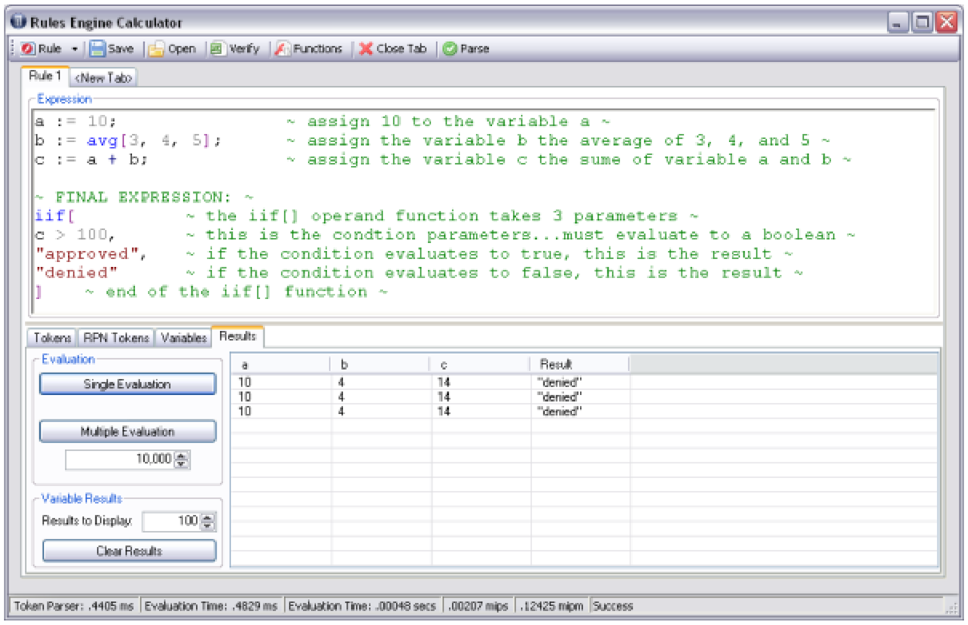
Variable	Value
a	Single Value = 10
b	Double click to set the variable value

Variable Name: b

The value for this variable is: 20

Token Parser: .0667 ms | Evaluation Time: 161.5894 ms | E

The evaluation engine allows you to add as much "white-space" as you need. That is, you can add tab characters and new lines; these will be ignored by the parser. Also, the Evaluation Engine supports comments. Comments are placed between ~ (tildas) and will be ignored. Note that comments can span multiple lines. Here's an example that uses assignments. Notice that the assignment operator is := and that each assignment must end with a ; (semicolon). The Rules Basic syntax dictates that the assignment declarations are made first, followed by the final expression that is evaluated. Note that the final expression that is evaluated can use any of the assigned variables. Also, assigned variables can use any previously declared assigned variables. Anything can be assigned to variables, including dates, string, numbers, Boolean, expressions, operand functions other variables, ... Here's a simple example (please read through the comments in the image for more details):



```
a := 10;                    ~ assign 10 to the variable a ~
b := avg[3, 4, 5];          ~ assign the variable b the average of 3, 4, and 5 ~
c := a + b;                 ~ assign the variable c the sume of variable a and b ~

~ FINAL EXPRESSION: ~
iif[             ~ the iif[] operand function takes 3 parameters ~
c > 100,         ~ this is the condtion parameters...must evaluate to a boolean ~
"approved",      ~ if the condition evaluates to true, this is the result ~
"denied"         ~ if the condition evaluates to false, this is the result ~
]     ~ end of the iif[] function ~
```

Multiple Evaluation: 10,000

Results to Display: 100

a	b	c	Result
10	4	14	"denied"
10	4	14	"denied"
10	4	14	"denied"

Token Parser: .4405 ms | Evaluation Time: .4829 ms | Evaluation Time: .00048 secs | .00207 mips | .12425 mipm | Success

Before I continue with a final example that is more realistic, let me give you the list of operand function that I programmed in the Evaluation Engine. If you need to add your own custom operand function....it's very simple but it is a code change to the Evaluation Engine. I thought long and hard if I should make the operand function external to the Evaluation Engine so that new operand functions could be added easily. The coding of the parser and evaluator was complicated enough so I decided to program the operand function as internal methods in the Evaluation Engine. Also, I wanted the Evaluation Engine to be as fast as I could make it so I thought that internal operand functions would be faster than having the Evaluation Engine invoke methods in external assemblies using reflection. Here's the list. The "Function" toolbar button also displays this list in the Rules Calculator:

Operand Function	**Syntax**	**Description**
Average	avg[p1, ..., pn] where p1,...,pn can be converted to doubles.	Calculates the average of a list of numbers. The list items must be able to convert to doubles.
Absolute Value	abs[p1] where p1 can be converted to a double.	Calculates the absolute value of a numeric parameter.
If-else-end	iif[c, a, b] where c is the condition and must evaluate to a Boolean. The value a is returned if c is true, otherwise, the value b is returned.	Performs an if-else-end
Lower Case	LCase[a]	Converts a string to lower case
Left	left[s, n] where s is the string and n is the number of characters	Returns the left number of characters from a string parameter.
Length	len[a] where a is a string variable	Returns the length of a string
Median	mid[p1, ..., pn] where p1, ..., pn are numeric values	Calculates the median for a list of numbers
Right	right[s, n] where s is the string and n is the number of characters.	Returns the right number of characters from a string parameter
Round	round[n, d] where n is the numeric	Rounds a numeric value to the number of decimal

Operand Function	Syntax	Description
	value to be rounded, and d is the number of decimal places.	places
Square Root	sqrt[a] where a is a numeric parameter	Calculates the square root of a number.
Upper Case	ucase[a]	Converts a string to upper case
Is Null or Empty	IsNullOrEmpty[a]	Indicates is the parameter is null or empty.
Is True or Null	isTrueorNull[a]	Indicates if the parameter has the value true or is null;
Is False or Null	IsFalseOrNull[a]	Indicates if the parameter has the value false or is null
Trim	trim[a]	Trims the spaces from the entire string
Right Trim	rtrim[a]	Trims the spaces from the right of a string
Left Trim	ltrim[a]	Trims the spaces from the left of a string
Date Add	dateadd[date, "type", amount] where date is a valid date, and type is "y", "m", "d" or "b" (representing year, month, day, or business days) and amount is an integer	Adds an amount to a date. Please note that the amount may be negative.
Concatenation	concat[p1, ..., pn]	This operand function concatenates the parameters together to make a string.
Date	date[m, d, y] where m is an integer and is the month, d is an integer and is the day, and y is an integer and is the year	Create a new date data type
Right Pad	rpad[a, b, n] where a and b are string values and n is numeric. The parameter p will be appended to the right of parameter a, n times.	Pads a string on the right with new values
Left Pad	lpad[a, b, n] where a and b are string values and n is numeric. The parameter p will be appended to the left of parameter a, n times.	Pads a string on the left with new values
Join	join[a, b1, ..., bn] where a is the delimiter and b1, ..., bn are the items to be joined.	Joins a list of items together using a delimiter

Operand Function	**Syntax**	**Description**
Search String	SearchString[a, n, b] where a is the string that is being searched, b is the string that is being sought, and n is the start position in a	Searches for a string within another string at a specified starting position
Day	day[d1] where d1 is a date value	Returns the day of a date
Month	month[d1] where d1 is a date value	Returns the month of a date
Year	year[d1] where d1 is a date	Returns the year of a date
Sub String	SubString[s, a, b] where s is the string, a is the starting point, and b is the number of characters extracted.	Extracts a substring from a string
Numeric Max	NumericMax[p1, ..., pn]	Finds the maximum numeric value in a list
Numeric Min	NumericMin[p1, ..., pn]	Finds the numeric minimum value in a list
Date Max	datemax[d1, ..., dn] where d1, ..., dn are dates	Returns the maximum date in the list
Date Min	datemin[d1, ..., dn] where d1, ..., dn are dates.	Returns the minimum date in the list.
String Max	StringMax[p1, ..., pn]	Finds the maximum string in the list
String Min	StringMin[p1, ..., pn]	Finds the minimum string in the list
Contains	contains[p1, p2, ..., pn] If p1 is in the list p2, ..., pn, this function returns "true" otherwise, this function returns "false".	Indicates if the item is contained in the list.
Between	between[var, val1, val2] where var, val1, and val2 are integers. if var >= val1 and var <= val2 then the function returns "true", otherwise, the function return "false".	Indicates if a value is between the other values. Please note that the comparison is inclusive.
Index Of	indexof[a, b1, ..., bn] If the list b1, ..., bn contains the value a, the index of the value is returned, otherwise, -1 is returned. Please note that this is zero based indexing	Returns the index of a list item.
Now	now[] This operand function takes no parameters	Returns the current date

Here's a slightly more realistic example of a rule that may be used by an insurance company to calculate the monthly payment of a life insurance policy. Here are the business rules for this life insurance policy:

- To qualify for this product, a "scoring" system is used. A customer attempting to qualify for this product is assigned an initial score of 100.
- If the customer is over 50 years of age, they loose 10 points. If the customer is over 70 years of age, they loose 40 points
- If the customer is a smoker, they loose 45.6 points
- If the customer's weight is more than 180 pounds, they loose 10 points. If the custom weighs more than 250 points, they loose 40 points
- The customer's final score is subtracted from 105.65. This amount is then multiplied by $5.25 to get the final payment.

The Rules Basic syntax for the above logic is saved in the file "InsPolicyA.rule". You can open this file in the Rules Calculator, set the values for the age, smoker, and weight and run the by clicking on the "Single Evaluate" button in the result tab. The final payment is calculate and returned as a string, such as "Monthly Payment = $374.00 (USD)".

Rules Engines are very powerful. In the above example, we have a complicated rule to calculate a customer's monthly payment for a product. A real insurance company probably has hundreds of products that all have special business rules. A financial institution, may have hundreds of products a customer can qualify for. A shipping company may have a set of business rules that dictate when items get shipped or re-ordered. If business rules are housed in a central location, administration of the rules becomes easier. If a rule changes, there is only one location to change the rules instead of all the client applications that embedded the rules in their logic. Also, if the Rules Engine is running on a Rules Server, there are additional benefits to be gained such as scalability, fault tolerance, and performs gains.

Client Application

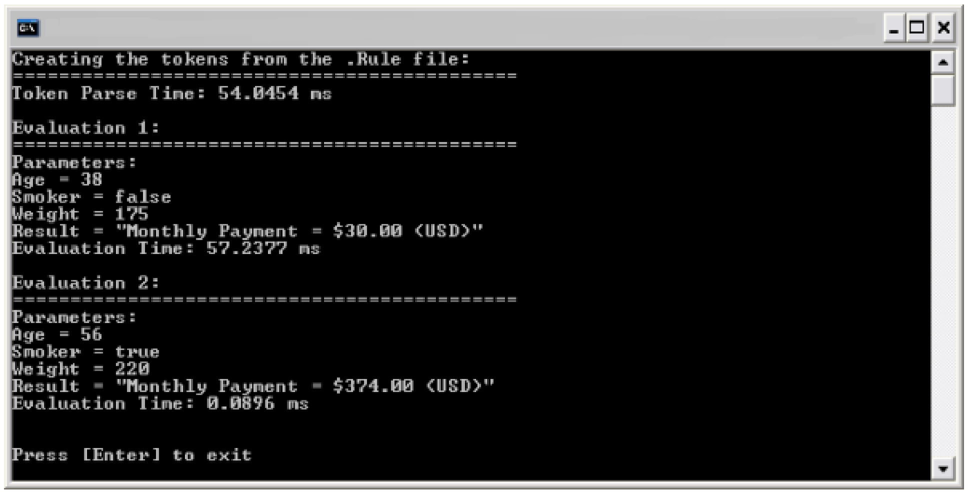
Creating a client application to consume the Evaluation Engine is a simple process. There are only 2 objects involved:

- EvaluationEngine.Parser.Token object. This object will allow you to open a saved .Rule file (or pass in a Rules Basic string) and set the values of the variables.
- EvaluationEngine.Evaluate.Evaluator object: This object takes the token object as a parameter in the constructor. The Evaluate() method does all the work to evaluate the tokens.

In the above insurance example, how could we evaluate that rule using a custom designed client? Here's a small console application that will load the rule, set the values for the variables and return the results:

```
// create the token object from the .Rule file
EvaluationEngine.Parser.Token token = new EvaluationEngine.Parser.Token(
     new System.IO.FileInfo("InsPolicyA.rule"));

// set the values for the variables
token.Variables["Age"].VariableValue = "38";
token.Variables["Smoker"].VariableValue = "false";
token.Variables["Weight"].VariableValue = "175";

// create the evaluator object and pass in the token object in the constructor
EvaluationEngine.Evaluate.Evaluator eval =
    new EvaluationEngine.Evaluate.Evaluator(token);

// run the evaluation
string ErrorMsg = "";
string result = "";
if (eval.Evaluate(out result, out ErrorMsg) == true)
     Console.WriteLine(result);
```

The download files contain a sample client console application. Here's a screen shot from that application:

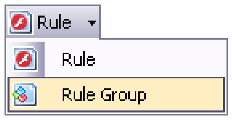


```
Creating the tokens from the .Rule file:
============================================
Token Parse Time: 54.0454 ms

Evaluation 1:
============================================
Parameters:
Age = 38
Smoker = false
Weight = 175
Result = "Monthly Payment = $30.00 (USD)"
Evaluation Time: 57.2377 ms

Evaluation 2:
============================================
Parameters:
Age = 56
Smoker = true
Weight = 220
Result = "Monthly Payment = $374.00 (USD)"
Evaluation Time: 0.0896 ms


Press [Enter] to exit
```

Rule Groups

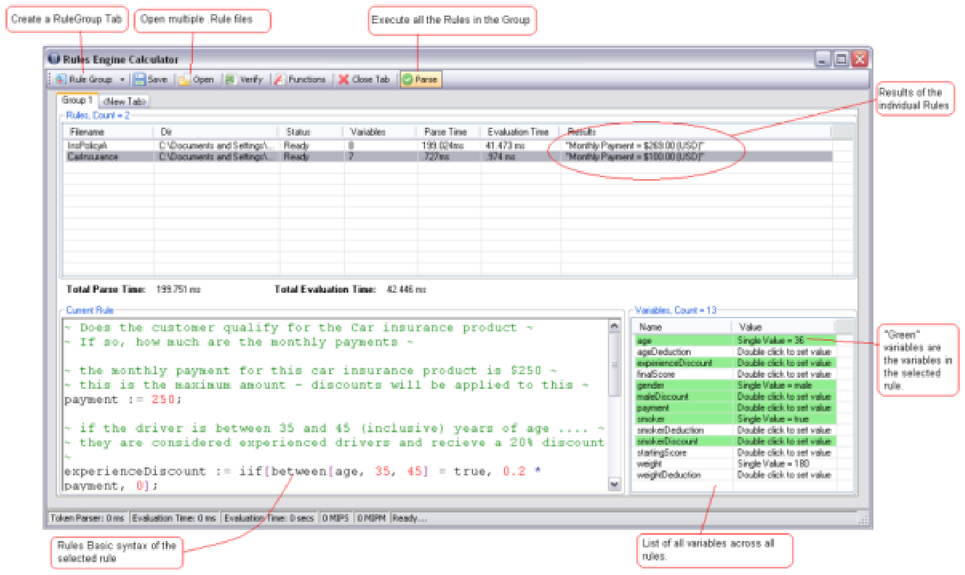
Assume that you create some rules with the Rules Calculator and you save the rules into .Rule files. Some of the variables that you have defined can be found in multiple rules. For example, consider the Insurance company again. We created a rule to see if a customer qualifies for a life insurance policy and that rules takes 3 parameters: age, smoker, and weight. The insurance company could also have another rule that provides rate quotes for car insurance. This car insurance rate quote rule could take 3 parameters: age, smoker, and gender. Therefore, amoung the 2 rules (Life Insurance Policy and Car Insurance) there are 4 parameters: age, smoker, weight, and gender. If we have these 4 parameters for a customer, it would be nice to run both rules. That's were the EvaluationEngine.Parser.TokenGroup object is used.

The Rules Calculator provides an implementation of the TokenGroup object. To add a new rule group in the Rules Calculator test application, click on the "Rule Group" in the drop down menu item:



This will create a new tab and load the Rule Group interface.

You cannot create a new rule in the Rule Group interface; You can only open saved .Rule files. Click on the "Open" toolbar button and open multiple .Rule files. After you have the Rules open, set the values for the variables by double clicking on the variables and setting their values. Finally, click on the "Parse" button in the main toolbar to execute all the rules. The results of the individual rules are displayed in the "Results" column of the List View.


Create a RuleGroup Tab
Open multiple .Rule files
Execute all the Rules in the Group
Results of the individual Rules
"Green" variables are the variables in the selected rule.
Rules Basic syntax of the selected rule
List of all variables across all rules.
Rules Engine Calculator
Rule Group
Save
Open
Verify
Functions
Close Tab
Parse
Group 1
<New Tab>
Rules, Count = 2
Current Rule
Variables, Count = 13

Filename	Dir	Status	Variables	Parse Time	Evaluation Time	Results
InsPolicyA	C:\Documents and Settings\...	Ready	8	199.024ms	41.473 ms	"Monthly Payment = $269.00 (USD)"
CarInsurance	C:\Documents and Settings\...	Ready	7	.727ms	.974 ms	"Monthly Payment = $100.00 (USD)"

Total Parse Time: 199.751 ms **Total Evaluation Time:** 42.446 ms

```
~ Does the customer qualify for the Car insurance product ~
~ If so, how much are the monthly payments ~

~ the monthly payment for this car insurance product is $250 ~
~ this is the maximum amount - discounts will be applied to this ~
payment := 250;

~ if the driver is between 35 and 45 (inclusive) years of age .... ~
~ they are considered experienced drivers and recieve a 20% discount
~
experienceDiscount := iif(between(age, 35, 45) = true, 0.2 *
payment, 0);
```

Name	Value
age	Single Value = 36
ageDeduction	Double click to set value
experienceDiscount	Double click to set value
finalScore	Double click to set value
gender	Single Value = male
maleDiscount	Double click to set value
payment	Double click to set value
smoker	Single Value = true
smokerDeduction	Double click to set value
smokerDiscount	Double click to set value
startingScore	Double click to set value
weight	Single Value = 180
weightDeduction	Double click to set value

Token Parser: 0 ms | Evaluation Time: 0 ms | Evaluation Time: 0 secs | 0 MIPS | 0 MIPM | Ready...

How does it work?

If you look at the source code, you will notice that I put a lot of comments in the code. The Evaluation Engine code has two major pieces: Parser and Evaluator. The Parser scans the Rules Basic syntax and creates the token collection. It then sorts the token collection in Reverse Polish Notation order. Finally, the sorted token collection is passed to the Evaluator object. I will describe each of these processes in detail.

The Parser

The parsing code is in the GetTokens() method of the Token object. This method starts a Do loop and begins extracting a character at a time from the RulesBasic syntax string. Each character is concatenated to a local string variable called "currentToken". Then the currentToken variable is analyzed and, depending on the analysis, a TokenItem object is created.

At any point in time, the parser knows exactly what it is looking for. For example, if the parser encounters a ~ character, it knows that it is starting a comment. Therefore, the parser knows that is can throw away (or dispose of) all of the characters it finds until the next ~ (closing comment indicator) is found. The parser manages its current state by using an enumeration (and a local variable) called ParseState. The parser only has to manage 5 states:

State	Description

State	Description
Parse_State_Operand	This state indicates that the parser is looking for an operand. Remember, the operand can be a variable, integer, string, Boolean, date, double, or null. This is the default value of the parse state; that is, the first thing the parser looks for when starting is an operand.
Parse_State_Operator	This indicates that the parser is looking for an operator. The operators can be mathematical, logical, or Boolean operators.
Parse_State_Quote	This indicates that the parser is looking for a double quote. Typically, when the closing quote is found a TokenItem object is created and set to data type string.
Parse_State_OperandFunction	This indicates that the parser has found a valid operand function and is parsing the parameters of the operand function. You will notice in the code that multiple parse states are maintained. For example, the current parse state indicates that we are parsing an operand function. However, a separate parse state is maintained to parser the parameters of the operand function. A separate parse state is needed for the parameters because the parameters to a operand function can also be valid expressions.
Parse_State_Comment	This parse state indicates that we are in a comment. Any characters that are encountered while in "comment-mode" are ignored. The parser is scanning the characters and looking for the ending comment indicator.

Now, with the help of parse state, the parser knows what it is looking for. For example, if the parser is looking for an operand and it encounters a "+" character (that is, it is looking for an operand and it finds an operator character), there may be a problem with the syntax since "+" characters are not allowed in operands. The point is that the parser knows what it is looking for and, of equal importance, it knows what it is looking for next. That is, if the parser is looking for an operand, then it knows that it will be looking for an operator next (or the end of the Rules Basic syntax).

There's a lot of code in the GetTokens() method so it's hard to detail it all in this article (I put a lot of comments in the code to help you understand what the parser is doing). After the parser extracts a character, you will see some nested switch statements and switch statements with log if-elseif statements. This is how I programmed the parser to know what it is currently looking for and what it is looking for next. Here's some code from GetTokens() with some detail around the space and comma handling. Notice that the switch statement looks at the current parse state. In the code sample below, if we are looking for a operand and we find a space, a token is created and the next parse state is set to look for a operator. Similarly, if we are looking for an operand and we find a comma and error is returned.

```
switch (parseState)
{
  case ParseState.Parse_State_Operand:
    #region Parse_State_Operand
    if (c == '"')
    {
      ....
    }
    else if (c == ' ')
    {
      #region Space Handling
      try
      {
        // we are looking for an operand and we found a space.
        // We are not currently looking for a closing quote.
        // Assume that the space indicates that the operand is completed
        // and we now need to look for an operator
        tokenCreated = CreateTokenItem(currentToken, false, false,
            out tempParseState, out isError, out lastErrorMessage);
        if (isError) return;

        if (tokenCreated == true)
        {
          // set the next parse state
          parseState = tempParseState;
          // reset the token
          currentToken = "";
        }
      }
      catch (Exception err)
      {
        lastErrorMessage = "Error in GetTokens() in
            Operand Space Handling: " + err.Message;
        return;
      }

      #endregion
    }
    else if (c == '(')
    {
      ....
    }
    else if (c == ')')
    {
      ....
    }
    else if (c == '[')
    {
      ....
    }
    else if (c == ']')
    {
      ....
    }
    else if (c == ',')
    {
      // we should never be looking for an operand and find a , (comma)
      lastErrorMessage = "Error in Rule Syntax:
           Found a , (comma) while looking for an operand.";
      return;
    }
    else if (c == '-')
    {
      ....
    }
    else if (c == ':')
    {
      ....
    }
    else if (c == ';')
    {
      ....
    }
    else
    {
      ....
    }
    #endregion
    break;

  case ParseState.Parse_State_OperandFunction:
    #region Parse_State_Operand
    ....
    #endregion
    break;

  case ParseState.Parse_State_Operator:
    #region Parse_State_Operand
    ....
    #endregion
    break;

  case ParseState.Parse_State_Quote:
    #region Parse_State_Operand
    ....
    #endregion
    break;
}
```

Creating operand tokens is a little more difficult then finding a space character and creating a operand token such as the case with this syntax: "1 + 1". It's more difficult because spaces are not required in the Rules Basic syntax. That is, "1+1" is a valid expression that does not contain any spaces. In this scenario, every time a character is appended to the string currentToken, the parser checks if the currentToken string variable contains an operator at the end. For example, when currentToken = "1+" the CreateTokenItem() method will find that the currentToken string ends with an operator. Furthermore, the CreateTokenItem() gets a little more complicated because it needs to "peek ahead". To understand what I mean, consider this example: "1>=2". Both the ">" and ">=" are valid operators. So, when the currentToken string variable is currently "1>", the CreateTokenItem() method needs to wait for the next character (the =) before the operand and operators are created. Take a look at the CreateTokenItem() to see how this is accomplished.

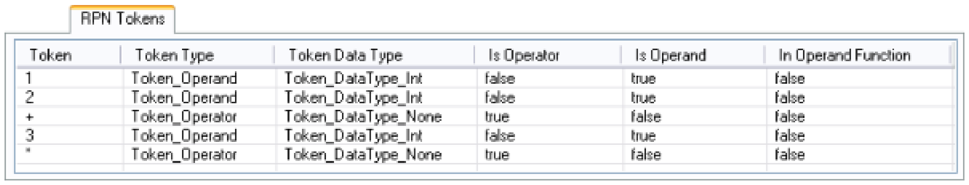
Sorting and Evaluating the Tokens

It's really easy for a human to look at the math expression "1 + 2 * 3" and know that the answer is 7 (using order of operations). It's difficult for a computer to evaluate this same expression. The computer needs to re-arrange the items/tokens in the expression so that it's easier to evaluate. In fact, the computer algorithm should re-arrange the expression so that it reads "2 3 * 1 +". This tells the computer that 2 and 3 need to be multiplied first and then 1 needs to be added to the result. The human notation "1 + 2 * 3" is known as Infix notation and the sorted computer notation "2 3 * 1 +" is known as Reverse Polish Notation (RPN). A computer can easily evaluate RPN simply by pushing and popping items from a Stack Abstract Data Type. Here's the simple algorithm for evaluation RPN:

1. For each item in the RPN statement:
    - If the item/token is a number, push it on the stack
    - If the item/token is an operator, pop two items from the stack and perform the operations. Push the result of the operation back onto the stack.
2. After all the items/tokens from the RPN statement have been evaluated, the stack should have 1 item which is the final answer.

In our simple example, "2 3 * 1 +", the 2 and 3 would be pushed to the stack. Then the 2 and 3 would be popped from the stack and multiplied. The result 6 would be pushed back on the stack. Next, 1 would be pushed on the stack. Final, 6 and 1 would be popped off the stack, then added together, and 7 would be pushed back on the stack. That is our final answer.

Obviously, this is a huge over simplification. The Evaluation Engine has to deal with operand functions, order of operations, variables, variable assignment, and so on. Fortunately, there is a famous computer scientist that invented an algorithm to convert from Infix notation to Reverse Polish Notation. This algorithm is called the Dijkstra Shunting Yard algorithm. More information about the algorithm can be found on Wikipedia at http://en.wikipedia.org/wiki/Shunting_yard_algorithm My implementation of the Shunting Yard algorithm can be found in the Token.MakeRPNQueue() method. The Rules Calculator displays the RPN for each expression in the "RPN Tokens" tab. For example, the RPN for (1 + 2) * 3 is


RPN Tokens

Token	Token Type	Token Data Type	Is Operator	Is Operand	In Operand Function
1	Token_Operand	Token_DataType_Int	false	true	false
2	Token_Operand	Token_DataType_Int	false	true	false
+	Token_Operator	Token_DataType_None	true	false	false
3	Token_Operand	Token_DataType_Int	false	true	false
*	Token_Operator	Token_DataType_None	true	false	false

Notice that the parenthesis for (1 + 2) * 3 disappear in the RPN list. That's because the RPN sort the tokens in such a way that parenthesis are no longer needed. The order of operations in an RPN stack is accomplished by the sort order. Again, please review the Token.MakeRPNQueue() method if you want details on the sorting algorithm.

Adding Additional Operand Functions

In the last part of this article, I would like to go through the exercise of adding a new Operand Function to the Evaluation Engine. Let’s add the trigonometry function Cosine to the Evaluation Engine. The Cosine Operand function will take 1 numeric parameter. Here's the steps involved to add the new Operand Function:

1. Add the name of the operand function in the OperandFunctions string array in the DataTypeCheck object:

```
public static string[] OperandFunctions = { ..., "cos" };
```

2. Add a "case:" condition in the switch statment in the Evaluator.EvaluateOperandFunction() method:

```
case "cos[":
  try
  {
    success = Cos(Parameters, out Result, out ErrorMsg);
  }
  catch (Exception err)
  {
    ErrorMsg = "Failed to evaluate the operand function " +
     OperandFunction.TokenName.Trim().ToLower() +
     ": " + err.Message;
    success = false;
  }
  break;
```

3. Program the Evaluator.Cos() method:

```
private bool Cos(Parser.TokenItems Parameters, out Parser.TokenItem Result, out string ErrorMsg)
```

```
{
    // initialize the outgoing variables
    ErrorMsg = "";
    Result = null;

    // make sure we have at least 1 parameter
    if (Parameters.Count != 1)
    {
        ErrorMsg = "Error in operand function Cos[]:
          Operand Function requires 1 parameter.";
        return false;
    }

    // we can only take a Cos of an item that can be converted to a double
    if (Support.DataTypeCheck.IsDouble(Parameters[0].TokenName) == true)
    {
        double temp = Parameters[0].TokenName_Double;
        double cos_temp = Math.Cos(temp);

        Result = new Parser.TokenItem(cos_temp.ToString(),
            EvaluationEngine.Parser.TokenType.Token_Operand,
            EvaluationEngine.Parser.TokenDataType.Token_DataType_Double,
            false);
    }
    else
    {
        ErrorMsg = "Error in operand function Cos[]: Operand Function
         can only evaluate parameters that can be converted to
         a double.";
        return false;
    }

    return true;
}
```

4. If you want to provide help on the new Cos[] Operand function add a new "case:" condition in the switch statement in the DataTypeCheck.FunctionDescription() method:

```
case "cos":
  Description = "Calculates the cosine of a number.";
  Syntax = "cos[p1] where p1 can be converted to doubles.";
  Example = "cos[90] < 0";
  break;
```

That's all there is to it; you can now call the cos[] Operand Function in your Rules Basic syntax. If you add any Operand Functions to the Evaluation Engine, I would request that you post the Operand Function code and the description code to this article on Code Project and I will include the Operand Function in the original source.

Short-Circuit Evaluation

Many of the business rules that I write are in the form of "if-else" statements. For example, if a customer has a certain attribute, they get a certain score. If a driver has certain driving violations, their premium goes up by X%. If a medical claim is in a certain state/condition, then delay the payment. Since many of the rules are of this "if-else" type, I have tried to optimize the iif[] operand function. Recall that the iif[] operand function takes 3 parameters: iif[c, a, b]. If the condition c evaluates to true, the result is a, otherwise, the result is b.

The formal definition of short-circuit evaluation (taken from Wikipedia) is "denotes the semantics of some boolean operators in some programming languages in which the second argument is only executed or evaluated if the first argument does not suffice to determine the value of the expression". This is nothing new to us, we have been using short-circuiting for years...take a look at the msdn help for || and see that it states: "The conditional-OR operator (||) performs a logical-OR of its bool operands, but only evaluates its second operand if necessary"

Here's how short-circuiting works in the Evaluation Engine:

1. When the parser scans the RulesBasic syntax, it finds all of the iif[] operand functions that do not contain other iif[] operand functions. The parser then sets the following property: tokenItem.CanShortCircuit = true;
2. When the sorting algorithm places the tokens in RPN order, the tokens that are parameters to a short-circuited iif[] operand function are placed in their own RPN stack (if fact, the tokens are placed in 3 RPN stacks representing the condition parameter, the true parameter, and the false parameter).
3. When the evaluator object is evaluating the tokens, the object knows which operand functions are short-circuited because of the new tokenItem.CanShortCircuit property. When the evaluator object encounters one of these short-circuit tokens, it knows to go to the condition RPN stack instead of the general RPN stack. If the condition RPN stack evaluates to true, then the True RPN stack is evaluated, otherwise the False RPN stack is evaluated. Since this is a complicated process, I created a ShortCircuit class that encapsulates all of the above logic.

The most important syntax to remember to take advantage of short-circuiting in the Evaluation Engine is do not nest the iif[] operand functions. If you do nest the iif[] functions, only the inner-most iif[] operand function will short-circuit. Instead of nesting the iif[] statements, break them into single statements and assign their result to a variable.

Here's a simple example:

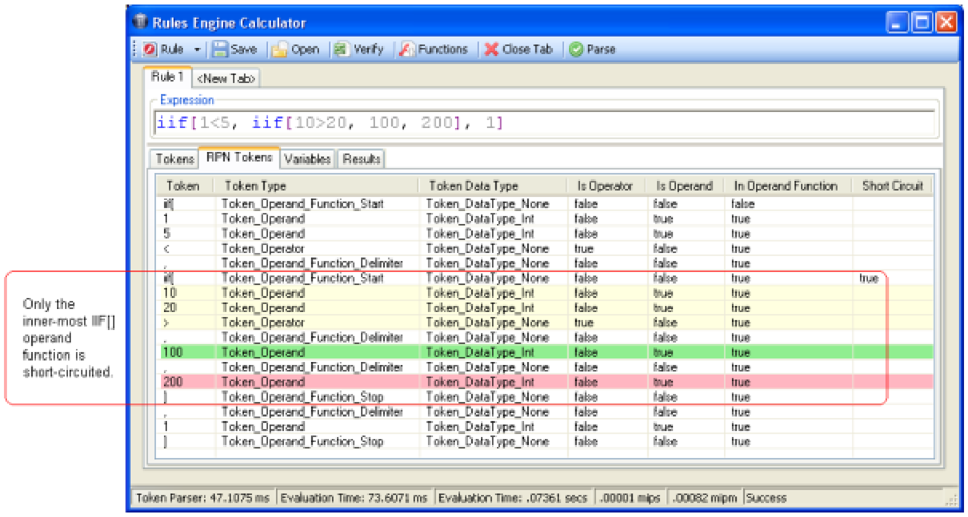
score := iif[10<5, (2 + 3) * 8 + (2 ^ 4) / 6, 1];

In the above example, the condition "10<5" will evaluate to false and the interpreter will immediately jump to the last parameter and return 1. That is, the true expression (2 + 3) * 8 + (2 ^ 4) / 6 will never be evaluated, thus saving you time.

I've noticed that many mathematical evaluators that include support for "if-else" functions will evaluate all the parameters before they perform the "if-else" statement. This causes problems with certain "if-else" statements such as:

iif[x = 0, 0, 100 / x], where x is an integer variable.

In the above example, if the evaluator does not support short-circuiting and the variable x is set to 0, many evaluators will return a "division by zero" error even though the "100 / x" does not need to be evaluated. Fortunately, the short-circuiting in the Evaluation Engine prevents us from having this problem.



```
iif[1<5, iif[10>20, 100, 200], 1]
```

Token	Token Type	Token Data Type	Is Operator	Is Operand	In Operand Function	Short Circuit
iif[	Token_Operand_Function_Start	Token_DataType_None	false	false	false	
1	Token_Operand	Token_DataType_Int	false	true	true	
5	Token_Operand	Token_DataType_Int	false	true	true	
<	Token_Operator	Token_DataType_None	true	false	true	
,	Token_Operand_Function_Delimiter	Token_DataType_None	false	false	true	
iif[	Token_Operand_Function_Start	Token_DataType_None	false	false	true	true
10	Token_Operand	Token_DataType_Int	false	true	true	
20	Token_Operand	Token_DataType_Int	false	true	true	
>	Token_Operator	Token_DataType_None	true	false	true	
,	Token_Operand_Function_Delimiter	Token_DataType_None	false	false	true	
100	Token_Operand	Token_DataType_Int	false	true	true	
,	Token_Operand_Function_Delimiter	Token_DataType_None	false	false	true	
200	Token_Operand	Token_DataType_Int	false	true	true	
]	Token_Operand_Function_Stop	Token_DataType_None	false	false	true	
,	Token_Operand_Function_Delimiter	Token_DataType_None	false	false	true	
1	Token_Operand	Token_DataType_Int	false	true	true	
]	Token_Operand_Function_Stop	Token_DataType_None	false	false	true	

Token Parser: 47.1075 ms | Evaluation Time: 73.6071 ms | Evaluation Time: .07361 secs | .00001 mips | .00082 mipm | Success

Conclusion

I'm sure there are many reasons why a developer releases his/her source code. My motivation for releasing this code is to solicit your feedback. The Evaluation Engine is an important part of a larger project that I am working on: the Rules Engine and Rules Server. If there is an error/bug in the Evaluation Engine, that error will "trickle" up to the other components that rely on the Evaluation Engine. If you like this library, please tell me. More importantly, If you don't like it, please let me know (I am open to constructive criticism). If you find a bug or have a suggestion, I would like that information too (I may make the change and re-release the code).

Thank You

The Rules Calculator application uses the control from the article Enabling syntax highlighting in a RichTextBox. Thank you Patrik.

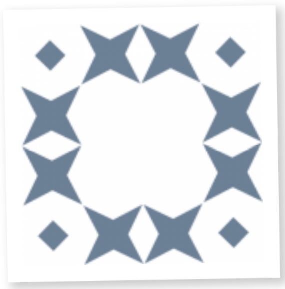
Revisions

Date	Description
May 23, 2008	Version 1 of the Evaluation Engine published on Code Project.
June 1, 2008	• Added a not[a] operand function • Added IsAllDigits[a] operand function • Added short-circuiting logic

License

License

About the Author



DonSn
Software Developer (Senior)
United States

Developer

Comments and Discussions

101 messages have been posted for this article Visit **https://www.codeproject.com/Articles/26314/Evaluation-Engine** to post and view comments on this article, or click **here** to get a print view with messages.