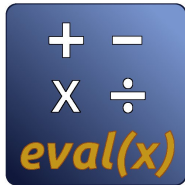


White Paper

rev 1.0
Feb 2nd 2015



Math Expression Evaluator for Android

Victor H Olvera

Contents

- 1 Program Overview
 - 1 Basic Description
 - 2 Operator Table
 - 3 Built in Functions Table
 - 4 Static Expressions
 - 2 Syntax Diagrams
 - 1 Recursive Decent Map
 - 2 Individual Syntax Diagrams
 - 3 Classes Overview
 - 1 tEvalScreen
 - 2 tParseTree
 - 3 tSyntaxNode
 - 4 Program Testing and Observations
 - 1 Test input/output
 - 2 Program behavior
 - 5 Further Reference
 - 6 Future Plans
-

1 - Program Overview

1.1 – Basic Description

The program takes in mathematical expressions and evaluates them. Its a little bit more useful then a common calculator in that it handles variables. The look and feel is more towards a programming language then a hand held calculator. The program is not sophisticated like MathCad and Linux octave, instead it was kept simple, as simple as possible.

The following is an example input:

```
apple = 3, pear = 4
10*(apple+pear*2)
```

The output should be:

```
110.0
```

The aim for this project was to explore basic design of computer languages. Using the program is straight forward hence its not covered in this paper. This paper is more of a technical reference guide then a users guide.

1.2 - Operator Table

Description	Usage	Precedence
+ Addition		4
- Subtraction		4
* Multiplication		3
/ Division		3
^ Exponent	$x \wedge y$ is x raised to the power y (right associative)	2
() Precedence grouping		1
= Variable assignment		5
+= Additive assignment	$x += y$ is equivalent to $x = x + y$	5
-= Subtractive assignment	$x -= y$ is equivalent to $x = x - y$	5
*= Multiplicative assignment	$x *= y$ is equivalent to $x = x * y$	5
/= Dividend-of assignment	$x /= y$ is equivalent to $x = x / y$	5
i= Divisor-of assignment	$x i= y$ is equivalent to $x = y / x$	5
, Expression delimiter	<expression>, <expression>, ...	6

1.3 – Built in Functions Table

Function	Description
random()	Returns a random number between 0 and 1
pi()	Returns pi with 15 digits precision
e()	Returns e with 15 digits precision
abs(x)	Absolute value of x
sqrt(x)	Square-root of x
cbrt(x)	Cube-root of x
exp(x)	e to the power of x
expm1(x)	(e to the power of x) - 1
ln(x)	Natural log of x
log(x)	Log base 10 of x
round(x)	x rounded to the nearest whole number
floor(x)	Floor of x
ceil(x)	Ceiling of x
cos(x)	Cosine of x
sin(x)	Sine of x
tan(x)	Tangent of x
acos(x)	Arc-cosine of x
asin(x)	Arc-sine of x
atan(x)	Arc-tangent of x

1.4 Static Expressions

Unfortunately the program does not support user functions but it does offer support for what I call a static expression. Static expressions work similar to lazy written functions. Say you want to work with the line formula $y = m \cdot x + c$. You can declare y to be a static expression and use it as a formula over and over again.

Say that in our situation $m = 1.4$, $x = 15$, and $c = 3$. Now you could use the line formula and determine y by entering the following.

```
y = m*x+c
```

what that does is evaluate $1.4 \cdot 15 + 3$ to 24.0 and assigns that value to y . That would work OK but its better to declare y to be static like in the next example. If you use the static keyword the program does not evaluate the expression right away, instead it saves it as a string and assigns that string to the variable.

```
static y = m*x+c
```

If you use `lsvars` to list out the variables. You would see that y is equal to the string " $m \cdot x + c$ ". You can still evaluate y by entering y by itself and pressing `[done]`. You can change the other variables m , x or c to determine other values of y without having to type the line equation again.

Below is the output of a clever example using static expressions and the `+=` operator

```
> m=1.5, x=0, c=3
1.5, 0.0, 3.0
> static y=m*x+c, x+= 1
m*x+b, x+=1
> y
3.0, 1.0
> y
4.5, 2.0
> y
6.0, 3.0
> y
7.5, 4.0
```

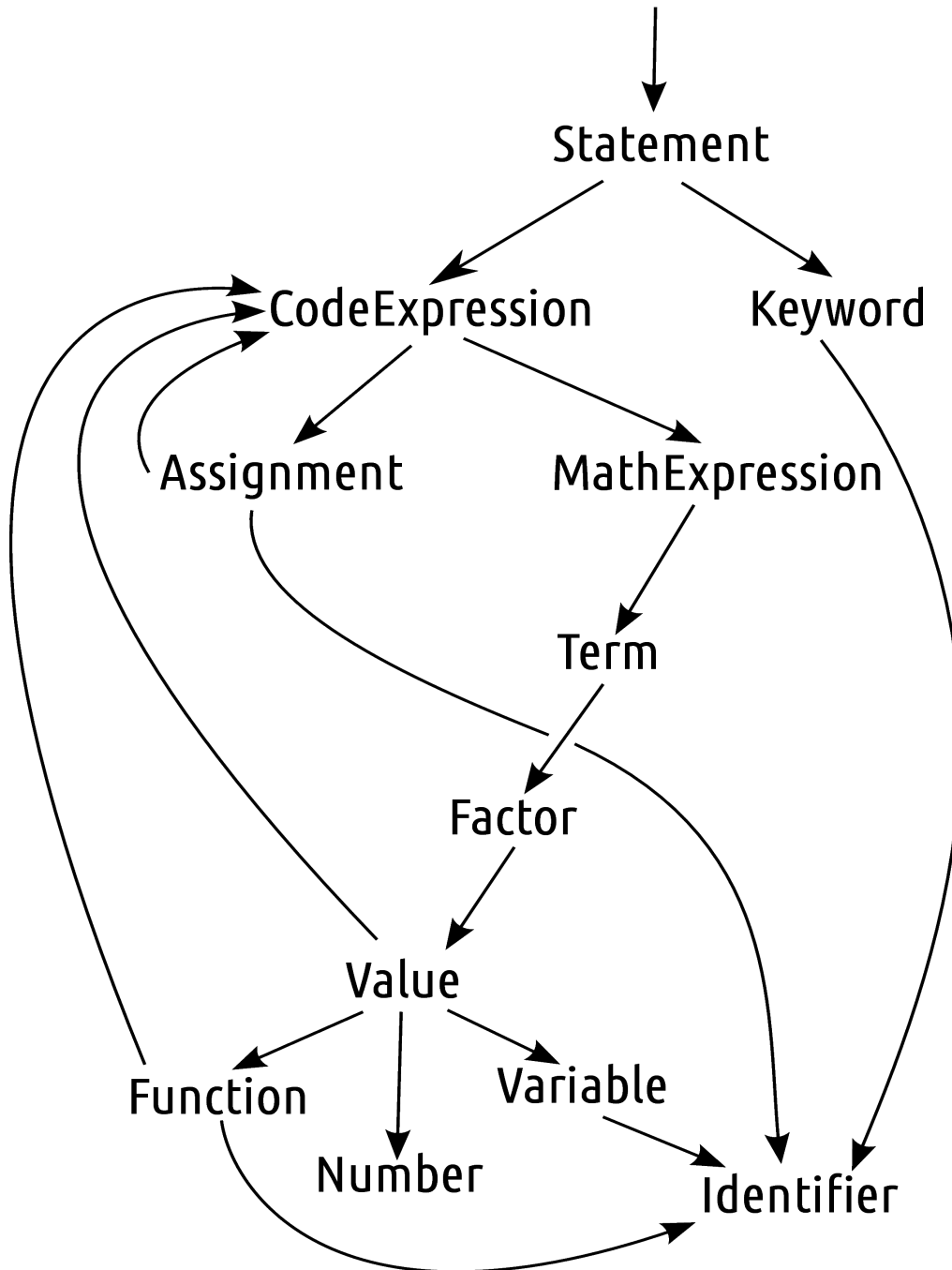
What that did was determine y when $x = 0, 1, 2, 3 \dots$. Every time the static expression y was executed, 1 was also added to x and that result is also printed.

Static expressions can be used as part of other expressions like regular variables (ie: $4 \cdot y + 1$ is valid if entered in the above example). When static expressions return multiple values only the first value is used for further computation.

2 - Syntax Diagrams

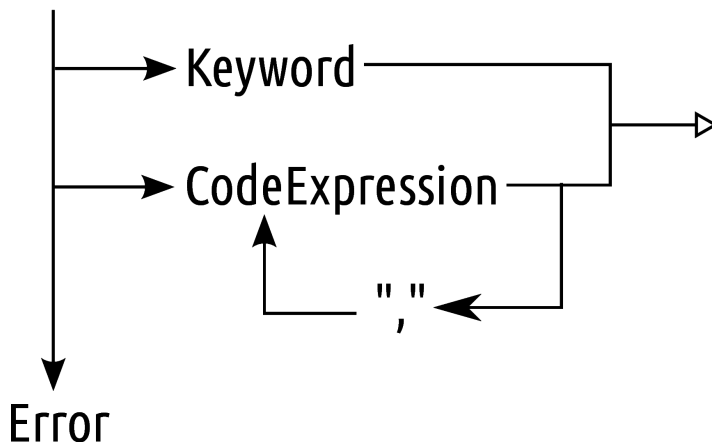
The program design relied heavily on syntax diagrams. The following Syntax Diagrams and the source code should mirror each other as much as possible.

2.1 - Recursive Decent Map

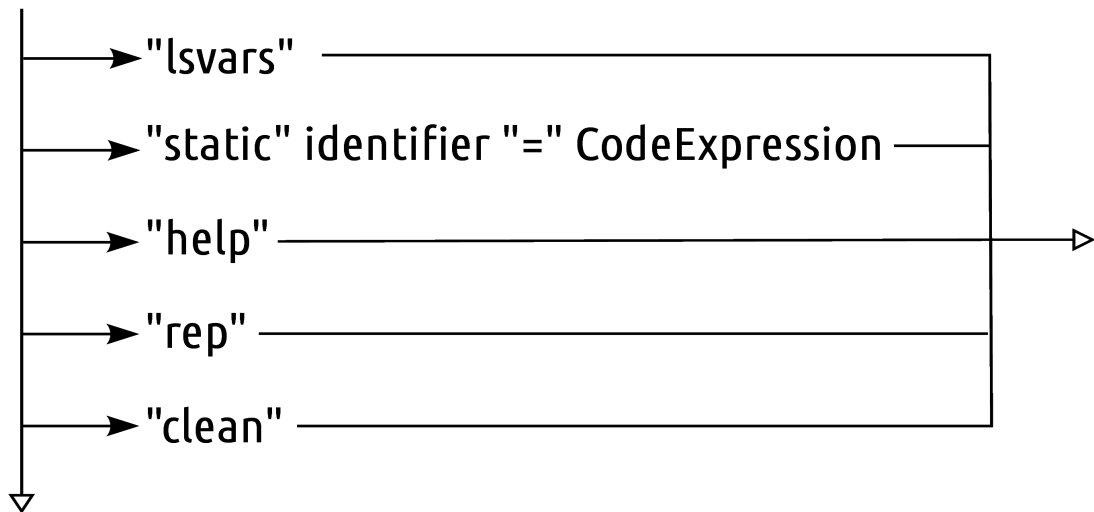


2.2 Individual Syntax Diagrams

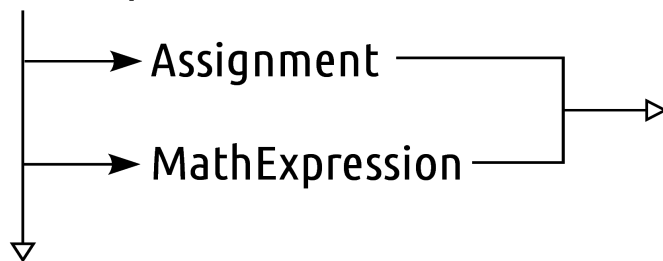
Statement



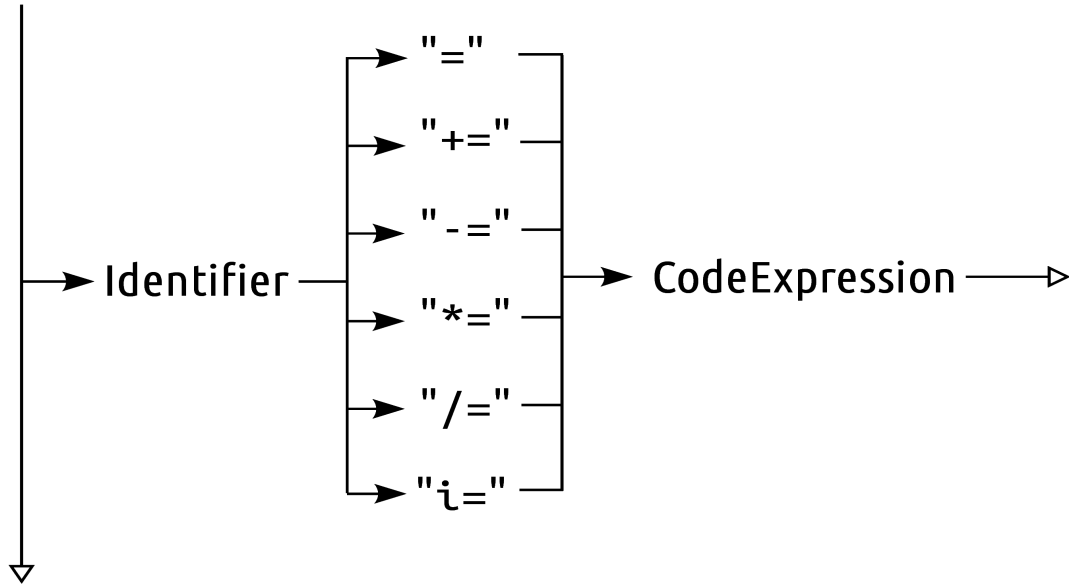
Keyword



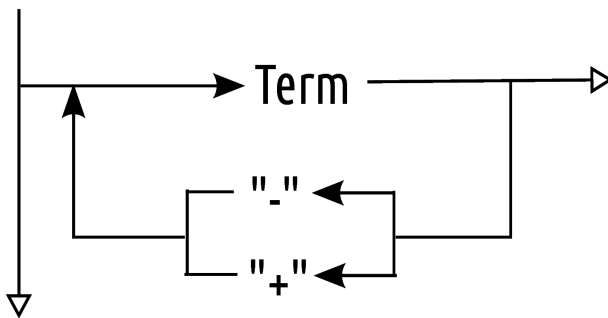
CodeExpression



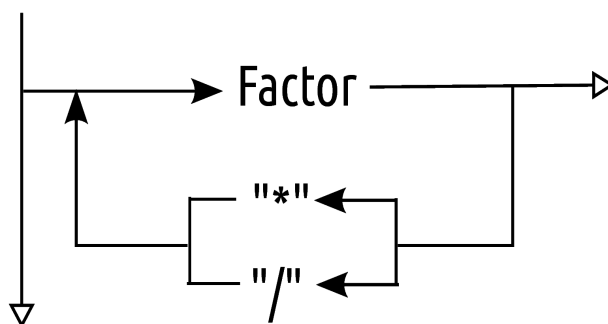
Assignment



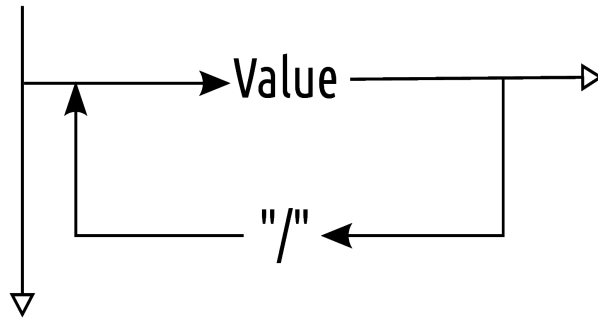
MathExpression



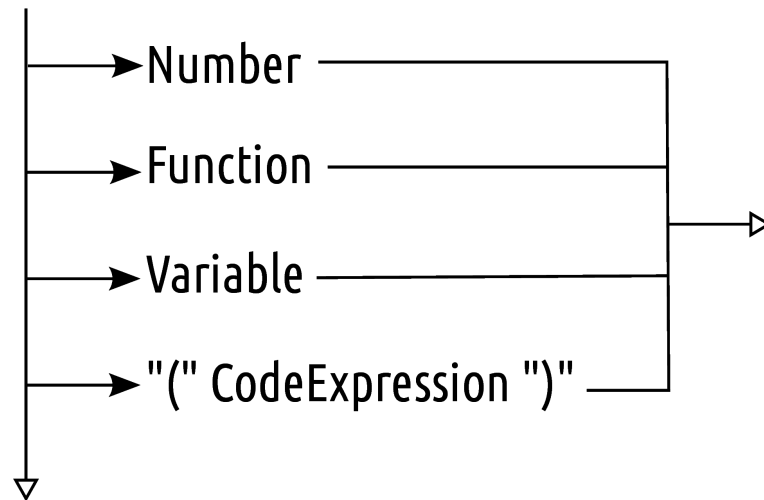
Term



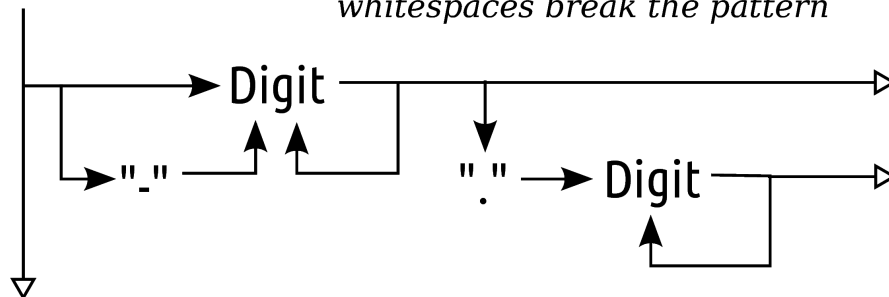
Factor



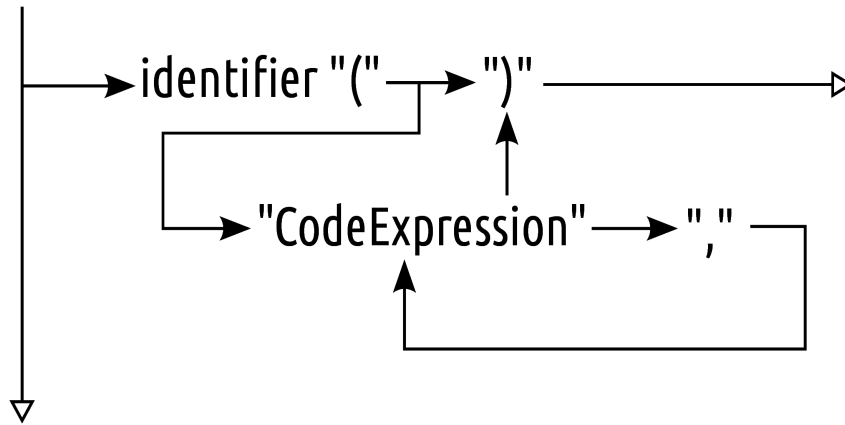
Value



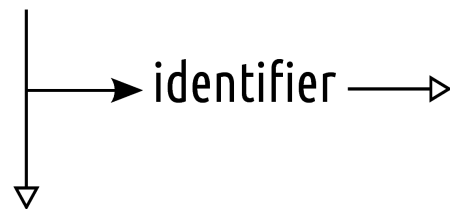
Number



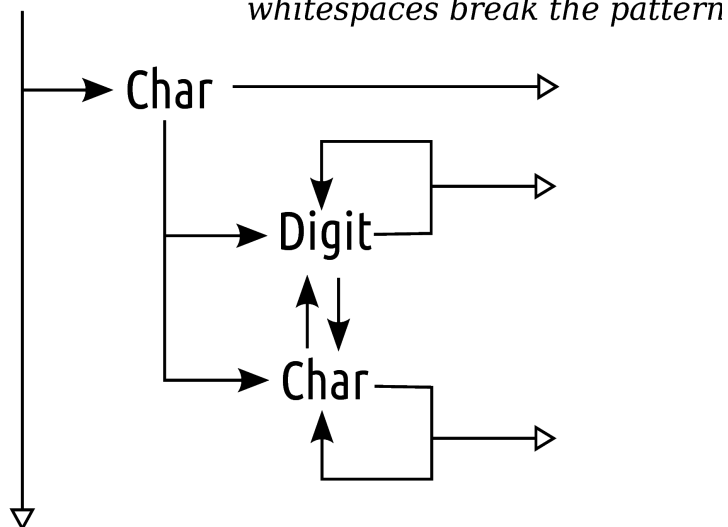
Function



Variable



Identifier



NOTE: a Char is any character between 'a' and 'z' or 'A' and 'Z' or '_'

3 - Classes Overview

3.1 – tEvalScreen

tEvalScreen is the GUI front for the program. It is simple, it only maintains a `ListView` for output and an `EditText` for input. A line of input is accepted by pressing the `[Done]` button and its passed on to a `tParseTree` object. The input is processed by the `tParseTree.parse(String)` method. That creates a syntax-tree which is stored internally within the `tParseTree` object. The syntax-tree is then evaluated by using the `tParseTree.eval()` method. The output of the `eval()` method is converted into a string and stored as an entry in the `ListView`.

Notetable identifiers within tEvalScreen:

- **onCreate()** handles initialization code
- **onEditorAction()** handles the input and output code
- **entry_list** JAVA list used to store the output as strings.
- **pt** maintains the syntax tree (`tParseTree`)
- **entry_count** keeps the number of items in the `entry_list`
- **R.id.entry_list_view** resource identifier of the `ListView`
- **R.id.entry_text_view** resource identifier of the `EditText`

NOTE: The program does not save its state when the screen is rotated.

Simplified view of the tEvalScreen class

```
public class EvalScreen extends Activity implements OnEditorActionListener {
    java.util.List<String> entry_list;    // used to store the output
    tParseTree pt;                       // maintains the syntax tree
    int entry_count;                     // number of items in the entry_list

    @Override
    protected void onCreate(Bundle savedInstanceState);

    @Override
    public boolean onEditorAction(TextView v, int actionId, KeyEvent event);
}
```

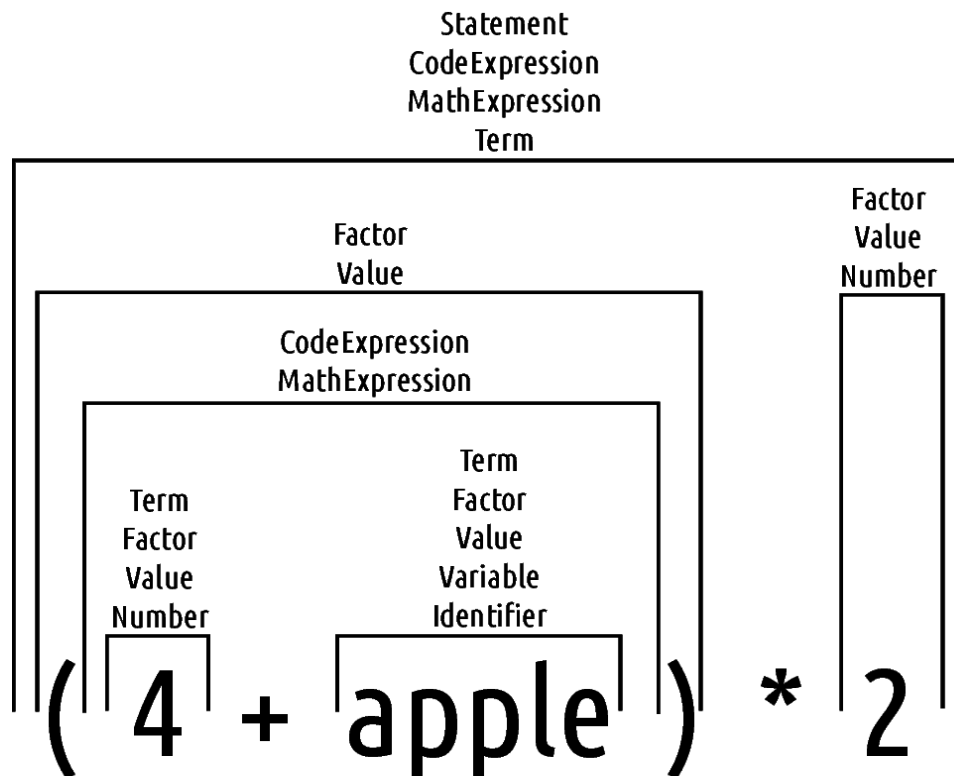
3.2 - tParseTree

The job of `tParseTree` is to take in text input, parse it, create a syntax-tree and evaluate it. `tParseTree` uses recursive decent (aka. divide and conquer) to break down syntax from a general view (represented by a statement) to a more simple view (such as a numeric constant or an identifier). See section 2.1

The methods in `tParseTree` can be categorized into three groups: constructors, interface, and parsing methods. The constructors and interface methods are for use by the GUI front-end. There are only two interface methods to worry about: `parse(string)` and `eval(node)`. `parse(string)` creates the syntax-tree from text and `eval(node)` is used to evaluate it. The parsing methods are prefixed by the word “Grab” hence they are referred to as the “grab” methods. The “grab” methods do all the heavy lifting for `parse(string)`. Each one of them is represented by a syntax diagram in section 2.2.

The job of a “grab” method is to recognize syntax from the input string and return a syntax tree. `GrabStatement()` should be able to recognize the syntax of any valid input. `GrabStatement()` achieves this by delegating its work to other methods which in turn delegate the job further until its broken down into something that is straight forward and easily recognizable. This form of parsing is called recursive decent. As each of the “grab” methods return; when successful; they return a piece of syntax tree and the number of characters it processed within the input string.

The figure below is a stack trace of the path that was taken while breaking down the input “(4+apple)*2”



tparseTree contains two embedded classes: tNodeCarrier and tVarStore. tNodeCarrier is used by the “grab” methods to return their work. A tParseTree object maintains its variables through a key/value dictionary of type Hashtable<String, tVarStore>. The String represents the identifier and the tVarStore represents the storage space.

Simplified view of the tParseTree class

```
public class tParseTree {
    public static final int tSTRING = 1;
    public static final int tDOUBLE = 2;

    class tNodeCarrier { // used by the “grab” methods to return their work
        tSyntaxNode node; // holds the syntax-tree
        int run; // number of chars processed
        // NOTE: if (run < 0) then node can't be evaluated (null or tERROR/tMISC)

        tNodeCarrier(tSyntaxNode node, int run);
    }

    class tVarStore { // storage type for variables
        int type; // either tSTRING or tDOUBLE
        String str; // valid if type == tSTRING
        Double num; // valid if type == tDOUBLE

        tVarStore(String st); // Creates a storage type tSTRING
        tVarStore(Double num); // Creates a storage type tDOUBLE
    }

    tSyntaxNode root; // holds the root of the syntax-tree
    Hashtable<String, tVarStore> var_table; // holds the active variables
    HashSet<String> fail_safe; // used to detect infinite static variables
    String previous_entry; // keeps the last entry; used by “rep” keyword
    String current_entry; // keeps current entry

    // constructors -----

    tParseTree()
    tParseTree(String st)

    // parsing methods -----

    // represents a single value; precedence 1
    tNodeCarrier GrabValue(String st);

    // handles all assignment operators; precedence 5
    tNodeCarrier GrabAssignment(String st);

    // represents a mathematical expression
    // handles '+' or '-' operators; precedence 4
    tNodeCarrier GrabMathExpression(String st);
```

```

// represents a term
// handles '*' or '/' operators; precedence 3
tNodeCarrier GrabTerm(String st);

// represents a factor
// handles '^' operator; precedence 2
tNodeCarrier GrabFactor(String st);

// handles a keyword
tNodeCarrier GrabKeyword(String st);

// represents a simple programming expression
tNodeCarrier GrabCodeExpression(String st);

// represents a single line of code
// handles the ',' delimiter
tNodeCarrier GrabStatement(String st);

// represents a numerical constant
tNodeCarrier GrabNumber(String st);

// represents a function call
tNodeCarrier GrabFunction(String st);

// represents a variable
tNodeCarrier GrabVariable(String st);

// represents an identifier
tNodeCarrier GrabIdentifier(String st, boolean filter);

// utility method used to skip white spaces
int CountWhiteSpaces(String st);

// interface methods -----

// NOTE: the syntax-tree is stored internally in root
// the tree is evaluated recursively
// calling eval() is the same as eval(root)

void Parse(String st);    // parses the string and creates a syntax-tree
tSyntaxNode eval();      // evaluates the whole syntax-tree
tSyntaxNode eval(tSyntaxNode node); // evaluates a node in the syntax-tree
}

```

Example usage of tParseTree

A tParseTree object can be initialized as follows:

an empty tree

```
tParseTree obj1 = new tParseTree();
```

a tree with some variables initialized

```
tParseTree obj2 = new tParseTree("x=1, y=10, a=b=c=0");
```

You can submit new input using the parse(String) method:

```
obj1.parse("apple = 4, pear = 17, grape = 71");
```

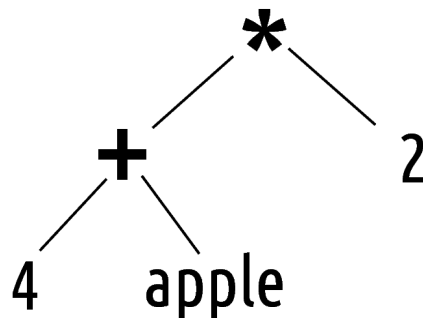
That would create a syntax-tree but not evaluate it. That means that the variables: apple, pear and grape would not be valid yet. Variables are not created until the syntax-tree is evaluated. Evaluation is done by calling the eval() method.

```
obj1.eval();
```

3.2 - tSyntaxNode

Every node within the syntax-tree is represented by the tSyntaxNode class. In mathematics everything is evaluated to a number. A syntax-tree can be described as a dependency tree. The leaves such tree would be entities that do not depend on anything for evaluation: for example a number or a variable. A node in the middle of a tree could be an operator such as "+". An operator depends on operands for evaluation. Hence in the case of "a+b", the variables "a" and "b" would be the leaves of the "+" node.

Below is an example of the syntax-tree generated by the input "(4+apple)*2". Please observe that parenthesis are not needed in a syntax-tree. Grouping and precedence is inherited within the location of a node.



NOTE: Unless its an error or special output from a keyword; any node should be evaluable into a floating point number.

Simplified view of the `tSyntaxNode` class

```
public class tSyntaxNode {
    public static final int cVALUE = 1; // node contains a double
    public static final int cIDENTIFIER = 2; // valid identifier
    public static final int cOPERATOR = 3; // math operator
    public static final int cFUNCTION = 4; // function call
    public static final int cVARIABLE = 5; // variable
    public static final int cPARSE = 6; // a parse-able string
    public static final int cMISC = 7; // text output from a keyword
    public static final int cERROR = -1; // error

    int type; // node type
    double value; // valid if type == cVALUE
    String ident; // valid if type == (cIDENTIFIER || cFUNCTION || cVARIABLE)
    String aux_msg; // valid if type == (cMISC || cPARSE || cERROR)
    int token; // valid if type == cOPERATOR

    // tree links
    tSyntaxNode parent = null;
    tSyntaxNode sibling = null;
    tSyntaxNode child = null;

    // constructors

    // initializes a cVALUE type
    tSyntaxNode(double value);

    // initializes a cOPERATOR type
    tSyntaxNode(int type, int token);

    // used to initialize any of the types that use "ident" or "aux_msg"
    tSyntaxNode(int type, String ident, String msg);

    // a somewhat smart version of the above constructor
    // can be used for cIDENTIFIER, cFUNCTION, cVARIABLE, cMISC, cPARSE, cERROR
    tSyntaxNode(int type, String st);

    // methods
    void AddChild(tSyntaxNode child); // appends a child to self ("this")
}
```

4 – Program Testing and Observations

4.1 – Test Input/Output

Test #	Description		
1	Simple whole number	in	4
		out	4.0
2	Simple floating point number	in	1.5
		out	1.5
3	Magnitude upper limit	in	9223372036854775807
		out	9223372036854776E18
4	Magnitude lower limit	in	-9223372036854775807
		out	-9223372036854776E18
5	Exceeding upper limit (magnitude)	in	9223372036854775808
		out	ERROR
6	Exceeding lower limit (magnitude)	in	-9223372036854775808
		out	ERROR
7	Precision limit	in	0.0000000000000000001
		out	1.0E-18
8	Exceeding precision limit	in	0.0000000000000000001
		out	0.0
9	A digit is required on the left side of the decimal point	in	0.
		out	ERROR
10	A digit is required on the right side of the decimal point	in	.0
		out	ERROR
11	Variable assignment	in	apple=2
		out	2.0
12	Accessing preinitialized variable apple = 2	in	apple
		out	2.0
13	Accessing uninitialized variable pear = <uninitialized>	in	pear
		out	ERROR
14	Initializing multiple variables	in	a=b=c=7
		out	7.0
15	Variable assignment using another variable apple = 2	in	pear=apple
		out	2.0

Test #	Description		
16	Unrecognizable character sequence	in	%&\$
		out	ERROR
17	Illegal variable name 1	in	or@ange=4
		out	ERROR
18	Illegal variable name 2	in	cindy.crawford=9
		out	ERROR
19	Illegal variable name 3 (identifier may not have a leading digit)	in	15cows=20
		out	ERROR
20	Valid variable name using upper/lower case letters, digits and “_”	in	_UPlow12=50
		out	50.0
21	Pressing [Done] with empty string	in	
		out	Repeats last entry
22	Entering the “lsvars” keyword	in	lsvars
		out	Shows variable listing
23	Entering the “rep” keyword	in	rep
		out	Repeats last entry
24	Entering the “help” keyword	in	help
		out	Shows a quick reference guide
25	Entering the “clean” keyword	in	clean
		out	All the variables should be deleted and “done!” should appear
26	Attempting to assign a value to a keyword	in	rep=23
		out	ERROR
27	Attempting to assign a keyword to a variable	in	apple=rep
		out	ERROR
28	Simple addition	in	2+2
		out	4.0
29	Simple subtraction	in	4-8
		out	-4.0
30	Subtracting a negative number	in	3--8
		out	11.0

Test #	Description		
31	Successive additions	in	1.3+2.5+3+4
		out	10.8
32	Mixed additions and subtractions	in	2-6+7.4-0.3-5
		out	-1.8999999999999995
33	Simple multiplication	in	4.3*3
		out	12.899999999999999
34	Simple division	in	2/0.5
		out	4.0
35	Successive multiplications	in	5*4*3*2*1
		out	120.0
36	Successive divisions	in	5/3/2/0.5
		out	1.6666666666666667
37	Simple use of the “^” operator	in	3^2
		out	9.0
38	Successive use of the “^” operator Note: The “^” operator is right-associative	in	2^3^2
		out	512.0
39	Complex expression	in	4+3*2-2*2^8-10/2
		out	14.0
40	Grouping parenthesis	in	(4+3)*2-2*2^(8-10)/2
		out	13.75
41	Assignment of a complex expression to a variable	in	apple=(3+7)/5
		out	2.0
42	Variables within a complex expression apple = 2 and pear = 8	in	(2+3)*apple-(2+pear/2)
		out	4.0
Note: For exhaustive testing, the next four tests should be performed on all functions			
43	Function call	in	sqrt(4)
		out	2.0
44	Too many parameters	in	sqrt(4,2)
		out	ERROR
45	Missing parameters	in	sqrt()
		out	ERROR

Test #	Description		
46	Improper parameters	in	<code>sqrt(\$#@)</code>
		out	ERROR
47	Variable assignment using a function	in	<code>area=4.7*pi()^2</code>
		out	46.38714068511998
48	Nested function calls	in	<code>cos(pi())</code>
		out	-1.0
49	Complex expression as parameter <code>apple = 2</code>	in	<code>cos((5+2)*2/apple*pi())</code>
		out	-1.0
50	Using “+=” operator <code>apple = 2</code>	in	<code>apple+=1</code>
		out	3.0
51	Using “-=” operator <code>apple = 3</code>	in	<code>apple-=1</code>
		out	2.0
52	Using “*=” operator <code>apple = 2</code>	in	<code>apple*=3</code>
		out	6.0
53	Using “/=” operator <code>apple = 6</code>	in	<code>apple/=3</code>
		out	2.0
54	Using “i=” operator <code>apple = 2</code>	in	<code>apple i= 1</code>
		out	0.5
55	Using “i=” operator without using spaces between variable and operator <code>apple = 0.5</code>	in	<code>applei=1</code>
		out	Variable apple should be unaffected and a new variable applei is created
56	Simple static variable	in	<code>static myexp=2+4</code>
		out	Should be listed as “2+4” in lsvars
57	Using static variable in other expressions <code>static myexp=2+4</code>	in	<code>myexp*4+7</code>
		out	31.0
58	Static variable with syntax error	in	<code>static myexp=2+3+</code>
		out	ERROR will be thrown after executing myexp
59	Static variable with uninitialized variables <code>grape = <uninitialized></code>	in	<code>static myexp=12*grape</code>
		out	ERROR will be thrown after executing myexp
60	Static variable with initialized variables <code>grape = 0.5</code>	in	<code>static myexp=12*grape</code>
		out	Executing myexp will yield 6.0

Test #	Description		
61	Recursively assigning a static variable to itself	in	<code>static myexp=myexp</code>
		out	ERROR will be thrown after executing myexp
62	Creating a race condition using static variables	in	<code>static a=b static b=a</code>
		out	ERROR will be thrown trying to execute either a or b
63	Entering multiple expressions	in	<code>2*5^2, 2+2</code>
		out	<code>50.0, 4.0</code>
64	Multiple expressions, initializing a variable and using it. Test #1 – left to right (x is uninitialized)	in	<code>x=2, x*3+8</code>
		out	<code>2.0, 14.0</code>
65	Multiple expressions, initializing a variable and using it, test #2 – right to left (y is uninitialized)	in	<code>y*3+8, y=2</code>
		out	<code>ERROR, 2.0</code>
66	Static variable with self-modifying variable <code>x = 0</code>	in	<code>static myexp=(x+=1)*2</code>
		out	Every time myexp is executed the next multiple of 2 should be shown, starting with 2.0
67	Static variable with multiple expressions and a self-modifying variable <code>x = 0</code>	in	<code>static myexp=x*2, x+=2</code>
		out	Every time myexp is executed two columns should appear, the first containing multiples of 4 the other multiples of 2, starting with 0.0, 2.0
68	Using a static variable with multiple expressions in other expressions <code>static myexp=1,2,3,4</code>	in	<code>myexp*2</code>
		out	<code>2.0</code> NOTE: Only the first value is used for computations. In this case 1*2=2.0

4.2 – Program Behavior

NOTE 1: Errors are not straight forward

The errors reported are not always useful because that requires extra code. The “grab” methods work as a gateways or masks that only allow valid syntax to go thru and for the most part they don't pay attention as to why did a particular text failed as valid syntax. To be able to do that requires extra code and that's beyond the scope of this version.

NOTE 2: Dangling characters found - explanation

Most of the time when a “grab” method fails it does not report a syntax error. As explained in note 1, they adopt a hands-off approach when dealing with errors. In general this approach causes a lot of permissiveness with the input. Take the following example:

```
pi() = 49.7
```

In this case `pi()` is a function call and can't be assigned a value. The error should have been caught by `CodeExpression()` but it's not because “`pi()`” by itself represents a valid expression and that's what it returns. `CodeExpression()` does not look ahead to check if you are trying to assign a value to it. The text following the call “`= 49.7`” is simply left unprocessed. In the end this type of error is caught by whom-ever calls `GrabStatement()` because when `GrabStatement()` returns all the input text should have been processed. That's where the “dangling characters found” error comes from.

NOTE 3: `pi = pi()` is a valid assignment.

Identifier name space is different between variables and functions; they don't cause conflict. Functions are detected by the parenthesis “`(...)`” after their identifier. Here the variable `pi` can coexist with the function `pi`.

NOTE 4: Infinite expansion of static variables

Trying to execute any of the following static variables would yield an error:

```
static x = x
static apple = pear
static pear = apple
```

They yield an error because they contain a string that expands on itself infinitely. This type of infinite expansion is caught using the `fail_safe` object during a call to `eval()` under the `cPARSE` section.

NOTE 5: Variable creation

Normal variables are created by the `eval()` method under the `cOPERATOR` section, `'=` token. Static variables are created by the `GrabKeyword()` method under the “static” if/then statement.

5 – Further Reference

Wikipedia Entry – Syntax Diagrams

http://en.wikipedia.org/wiki/Syntax_diagram

Wikipedia Entry – Abstract Syntax Tree

http://en.wikipedia.org/wiki/Abstract_syntax_tree

Wikipedia Entry – Recursive Descent

http://en.wikipedia.org/wiki/Recursive_descent_parser

Compiler Construction – Niklaus Wirth (1996/2005)

<http://www.ethoberon.ethz.ch/WirthPubl/CBEAll.pdf>

6 – Future Plans

Personally I am interested in writing a script language that can handle user defined functions, branching and loops. Hence I'll be adding those features to this program. That's what I have planned for version 2.0

A cool way to add international support is to modify `GrabIdentifier()` to allow it to support other alphabets such as Cyrillic. I don't see happening because I only know the Latin alphabet.

Adding support for plotting or graphing functions would not be too difficult. That would be a feature for version 3.0.