Paradox

the

Simple Engine

for

Parsing And Evaluating

Infix Mathematical Expressions

using

A Recursive Descent Parsing Algorithm

by Michael Lam

February 2003

Grade 11

Introduction................................4

Goals & Expectations..........................4

Description of Methods.......................5

History.......................................5

Predevelopment..................................5

Calc1..6

Calc2..6

Calc3..6

Calc4..6

Paradox.....................................7

Theory..7.

hterface.................................25

pts & Scripting............................26

usion................................28

ential Applications.......................28

ure Additions & Enhance

PARADOX - A SIMPLE ENGINE FOR PARSING AND EVALUATING INFIX MATHEMATICAL EXPRESSIONS USING A RECURSIVE DESCENT PARSING ALGORITHM

Michael Lam
Lam Home School, 20002 Wooded View Lane, Elkton, VA 22827

Evaluating a mathematical expression such as "$2 + 1.05e\text{-}18[5^3(\sin 4)(\quad -7 \log_2 3)]$" is a unique challenge, even for a computer. Much of the challenge is associated with the act of cutting or "parsing" the actual text of the expression into a form that computers can understand. One method of doing this is with a recursive descent parsing algorithm, which builds an expression tree by recursively parsing an expression according to a special grammar.

Paradox is a set of Java classes I programmed with JDK v1.4.1 that will parse and evaluate a mathematical expression such as the one written above. It supports variables, functions, boolean operators, subexpressions, and developer-defined functions. I also created Calc4, a helper application. It provides a graphical user interface (GUI) for the Paradox engine, allowing the user to type in complex expressions and obtain the numerical answer. Calc4 also demonstrates the benefits of using a simple scripting language to automate routine calculations. Altogether, Paradox is a reliable, expandable engine that has many potential uses at the home, on the job and in the classroom.

Introduction

Evaluating a complex mathematical expression such as "$2 + 1.05e\text{-}18[5^3(\sin 4)(\quad - 7 \log_2 3)]$" is a unique challenge, even for a computer. The main problem is that the computer cannot go directly from the textual representation of an expression to a form that it can calculate the answer to. Much of the challenge, then, is associated with the act of cutting or "parsing" the actual text of the expression into a more workable form, such as an expression tree. Once an expression tree has been built, the computer can easily obtain a numerical answer.

Goals & Expectations

The goal of this project was two-fold: 1) to write an engine (ie. an API – Application Programming Interface) which would solve the problem mentioned above, facilitating the parsing and evaluating of infix mathematical expressions and 2) to write a sample front-end GUI that would provide a "nice" interface to the functions of the engine.

The expectations for the Paradox engine were as follows:

- That it be able to reliably parse an expression string into an expression tree so that the expression could be evaluated multiple times without re-parsing.
- That it be able to provide a string representation of an expression tree.
- That it be able to handle all common operators plus boolean operations and exponents.
- That it be able to recognize scientific notation in some form.
- That it be able to handle subexpressions, implied multiplication and extra whitespace in expressions.
- That it allow the user to use variables and constants with multi-character names to hold temporary values and mathematical constants, respectively.
- That it provide many basic functions as well as the ability to add more later.
- That it provide intelligent and useful error messages when an error does occur.
- That it be packaged into an easy-to-use API for any developer to use in their program to provide expression parsing and evaluating capabilities.

The expectations for the Calc4 front-end application were as follows:

- That it provide a functional and aesthetically pleasing multiline display for entering expressions and obtaining numerical results.
- That it provide an area for reporting errors to the user.
- That it provide a full-featured set of on-screen buttons for those who prefer to use the mouse to enter expressions.
- That it provide the option to show on-screen representations of expression trees.
- That it implement a simple scripting language to demonstrate the possibilities of such a device for performing routine calculations.
- That the interface be simple and easy-to-use as well as logical and functional.

Description of Methods

The Paradox engine and the Calc4 interface program were programmed in Java over the course of several months, mainly using a 266 Mhz Compaq laptop with 128 MB RAM running Windows '95 (upgraded to Windows '98 part of the way through the project). The project was started using the Java Development Kit (JDK) v1.3.1, but was switched to the JDK v1.4.1 when the new version was released. The Xinox JCreator Lite Edition Integrated Development Environment (IDE) was also used as a front-end to the JDK.

History

Predevelopment

During my AP Computer Science class last year, I became rather fascinated by binary trees and by the problem of coding a routine to evaluate infix mathematical expressions. Evaluating prefix and postfix expressions were easy to evaluate with stacks; infix expressions, however, posed more of a problem since you couldn't simply push and pop operators and operands using a stack. My AP Computer Science textbook gave the following algorithm for parsing infix expressions:

```
If the expression is empty, do nothing.
If the expression is a single operand, create one leaf node for
   that operand.
If the entire expression is enclosed in parentheses, drop them
   and parse (recursive step) the expression inside.
Otherwise, find the last operator of the lowest precedence order
   outside of any parentheses.  Create a root node for that
   operator.  Parse (recursive step) the part of the expression
   to the left of that operator and append the resulting tree as
   the root's left subtree.  Parse the part of the expression to
   the right of the operator and append the resulting tree as the
   root's right subtree.[1]
```

Using C++ and MFC (Microsoft Foundation Classes, a library for creating interfaces in Windows), I programmed a simple calculator application using that algorithm. It supported all the basic operators (including the modulus and power operators), parenthetical expressions, scientific notation and numerous functions. It could also store 26 variables ("a" to "z"). However, the program was unreliable and the project soon became what programmers refer to as “spaghetti-code,” meaning that the code was hard to understand and hard to debug. Since I had no idea how to proceed from there anyway, I set the project aside for a time.

I began to search for an alternate algorithm for evaluating an infix mathematical expression. I did numerous searches on the internet, both through public search engines and the local community college's InfoTRAC system.

After a long search, I finally found an alternate algorithm that seemed promising: the recursive-descent parsing algorithm. Using this algorithm, I could combine my desire to program an infix expression calculator with my fascination with binary trees (or, in this case, expression trees).

1 Litvin 437.

Calc1

The first incarnation of the Paradox engine was titled "Calc1," for lack of a better name. It was basically a much-expanded version of the algorithm presented on the Java Notes website[2]. It supported all the basic operators (including the modulus and power operator), parenthetical expressions, scientific notation and numerous functions. I created the CalcIn class for use as a specialized input stream, and exceptions were thrown using the ParseException class. This version was text-based and ran in a console window on windowed operating systems.

Calc2

Having most of the code for the engine together in one class might be convenient, but it quickly became overwhelming. I decided to separate everything out into classes for a more logical organization. Also, I had by this time found a much better grammar at the JEP website. Instead of trying to modify the previous Calc1 code, I decided to completely re-write the engine. This I did, and the results were invigorating. Since the parsing part of the engine simply built an expression tree and was separate from the evaluating part of the engine, an expression could be parsed once and evaluated multiple times. Like the previous version, Calc2 was text-based and ran in a console. Its list of supported features was long:

- Basic operators (+, -, *, /) plus modulus (%) and power (^) operators
- Boolean operators (==, !=, <=, >=, <, >, &&, ||, &, |, !) resulting in a value of either true (1) or false (0)
- Parenthetical expressions and sub-expressions
- Scientific notation with both 'E' and 'e'
- Two exception classes: ParseException and EvalException
- ParadoxEngine wrapper class with Parse(), Eval() and Evaluate() functions
- Numerous functions, with the ability to add more externally by overloading the Function class and calling the addFunction() method of an instance of the ParadoxEngine class
- Variables with names of any length, provided that they start with a letter
- Multiple assignment operators (=, +=, -=, *=, /=, %=, ^=)
- Many predefined constants and the ability to add more by either calling the addConstant() method of an instance of the ParadoxEngine class (developers) or by using the "#=" assignment operator (users)
- Variable "ans" for storing the results of the previous expression

Calc3

There was not much change in the actual engine between Calc2 and Calc3, but the nice windowed interface replaced the console input/output of the previous two versions.

Calc4

Calc4 was the second major change of the Paradox engine. In this version, the actual engine was separated from the user interface program. The engine code was stored in a package ("paradox") and the user interface program was a separate project that imported the package.

2 Eck, David J. <http://math.hws.edu/javanotes/c11/s5.html>

This allows for the easy distribution of the engine on the internet, while keeping the user interface a separate project.

Paradox

Although "Paradox" is the name of the entire project, it refers most specifically to the actual engine (or API). Just like a car engine is the part of the car that does most of the hard work, Paradox is the part of this project that does most of the hard work. Its job is to take care of all the details of parsing and evaluating mathematical expressions. The developer or user should be able to give it expressions such as "`2+(1.05e-4){5³sin(4)[pi-7*log2(3)]}`" and receive a numerical answer (2.04518) in return.

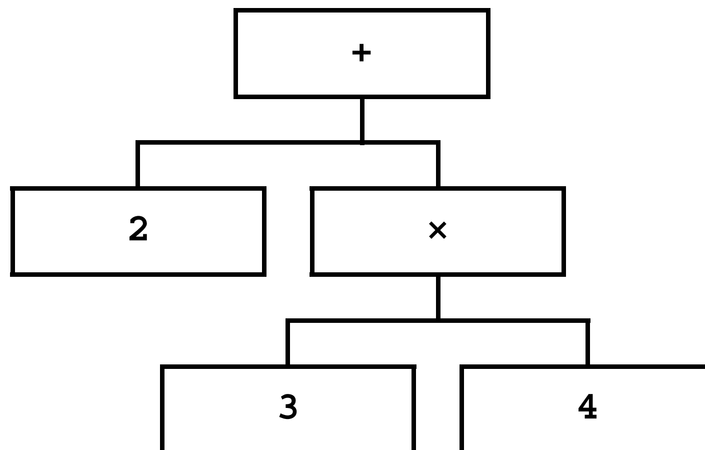
The following discussion concerning Paradox has been split into two parts, the first dealing with the theories and concepts behind the engine, and the second dealing with the actual Java implementation.

Theory

The main problem in this sort of program is converting an expression such as "2 + 3 × 4" into a form that the computer can use. "2 + 3 × 4" is considered a "character string" by the computer; that is, it sees it as simply a string (sequence) of characters (single letters, numbers or symbols). To understand the expression, the computer must build an "expression tree," first "parsing" the expression according to a defined "grammar." Each of these concepts is explained in detail in this section concerning the theory behind the Paradox engine.

Expression Trees

The most logical way to represent a mathematical expression on a computer is to use what is termed an "expression tree." In an expression tree, the different parts of an expression are organized in a tree-like structure. For instance, the expression "2 + 3 × 4" could be represented by the following expression tree:



In this tree, each box is called a "node" and the boxes directly under it are called its "subnodes." The node at the top of the tree (in this case, the "+" node) is called the "root" node. The nodes are the base of the tree that have no subnodes are called "leaf" nodes (in this case, the "2", "3" and "4" nodes). All of the nodes in between are called "branch" nodes. Any node that has a subnode is also referred to as a "parent" node, and any subnodes are also referred to as "child" nodes.

In the computer, each node is a series of bytes stored in an allocated chunk of memory. Each root or branch node contains pointers to the locations of its subnodes.

The concept of an expression tree is very similar to the concept of a binary tree, a concept that was explored extensively in my AP Computer Science course. The difference is that a node in a binary tree is limited to having two subnodes (thus the term "binary"), while nodes in an expression tree may have as many subnodes as necessary.

To evaluate the above expression tree, simply go from top to bottom, left to right starting at the root node. The root node is "+," thus its subnodes must be added together. It has two subnodes, one of which is a leaf node ("2"). The other is a branch node ("×"). The multiplication sign indicates that its subnodes must be multiplied together. Since both of its subnodes are leaf nodes, they can be immediately multiplied. This product is then added to the "2" node to obtain the final result.

This expression tree could also be expressed using the following notation:

```
Root
  Leaf (2)
+ Branch
    Leaf (3)
  × Leaf (4)
```

This is an extremely simple tree. Here is the expression tree that Paradox builds of the same expression ("2 + 3 × 4"):

```
Expression
  OrExpression
    AndExpression
      EqualExpression
        RelationalExpression
          AdditiveExpression
            MultiplicativeExpression
              UnaryExpression
                PowerExpression
                  UnaryExpressionNotPlusMinus (2)
          + MultiplicativeExpression
              UnaryExpression
                PowerExpression
                  UnaryExpressionNotPlusMinus (3)
            * UnaryExpression
                PowerExpression
                  UnaryExpressionNotPlusMinus (4)
```

Why is this tree so much more complicated than the first? Most of the complexity is caused

by the extensive grammar that Paradox uses, which actually makes the computer's job much simpler (grammars are covered in the next section).

Grammars & Backus-Naur Form

Before an expression can be parsed into an expression tree, it is helpful to define a "grammar." The grammar specifies how the different parts of an expression are denoted and provides clues as to how to construct a tree. In a grammar, there are "terminals" and "non-terminals." A terminal represents a leaf node of the expression tree, such as a number or a specific character. A non-terminal represents the root node or a branch node of the expression tree and may contain terminal and/or other non-terminals. A grammar could be stated in a verbose paragraph form, such as the following (note that in this paragraph, non-terminals are contained in quotation marks and terminals are in bold):

> An "expression" is defined as at least one but possibly more "term." A term is defined as at least one but possibly more "factor." A "factor" is either an "expression" surrounded by **parentheses** or a **number**.

Although this form is sufficient for simple grammars, a more concise and clear form is needed for more complex grammars. The most commonly used form is the Backus-Naur Form (usually abbreviated as BNF). BNF was mostly invented by John Backus in 1959 as a formal notation describing the syntax of ALGOL 58, a programming language. It was later modified by Peter Naur in 1960.[3] Since then, several different variations have arisen, such as Extended BNF, but they all share many common characteristics. BNF uses symbols called "meta-symbols" to abbreviate common phrases like "is defined as" and to represent ideas such as "at least one but possibly more." The following meta-symbols are commonly accepted in BNF:

```
::=     "is defined as"
|       "or"
<>      non-terminal
""      terminal character(s)
[]      optional item(s)
{}      0, 1, or more item(s)
```

It is also common to drop the "<" and ">" characters around non-terminals and to put non-terminals that are not single characters in bold or between asterisks (*). Using BNF, then, the grammar described by the paragraph above could be defined as follows:

```
Expression  ::=  Term { Term }
Term        ::=  Factor { Factor }
Factor      ::=  Number |
                 "(" Expression ")"
```

As a matter of interest, the definition of BNF itself can be expressed in BNF:

```
Syntax        ::=  { Rule }
Rule          ::=  Identifier "::=" Expression
Expression    ::=  Term { "|" Term }
```

3 <http://cui.unige.ch/db-research/Enseignement/analyseinfo/AboutBNF.html>

```
Term           ::=  Factor { Factor }
Factor         ::=  Identifier  |
                    QuotedSymbol |
                    "(" Expression ")" |
                    "[" Expression "]" |
                    "{" Expression "}"
Identifier     ::=  Letter { Letter | Digit }
QuotedSymbol   ::=  """ { Character } """
```

The grammar used by Paradox is based on a grammar[4] used by a similar project called the Java Math Expression Parser (or JEP). I made a few modifications, most of which involved adding extra operators to allow the user more choices in the process of entering expressions. I also modified the Expression class to allow for assignments. The final grammar used by Paradox is shown in Appendix A.

Recursive Descent Parsing

Now that expression trees have been explained and the need for a defined grammar shown, all that remains is to finish the discussion by describing the parsing technique used by Paradox. "Parsing" is the act of cutting up a character string into different parts. There are many ways of parsing a string. One of the simplest ways is to read through the string, making a "cut" whenever a certain character (called the "delimiter") is found. For instance, the string "this is a string" parsed with a space as the delimiter would result in four separate strings: "this", "is", "a", and "string".

This method is sufficient for simple parsing, but when attempting to parse a complex mathematical expression, this method is utterly inadequate, and a more sophisticated method is needed. The Paradox engine uses a technique called "recursive descent parsing"—sometimes referred to as (LL)1 parsing—to achieve its goals. Recursive descent parsing is the tool that allows the computer to convert a character string into an expression tree, from which it can obtain a numerical answer. Although it is actually a fairly common type of parsing, it is very powerful and can actually be used to parse a programming language. A recursive descent grammar for the Java language itself can be seen in Appendix C.

Before recursive descent parsing can be understood, the concept of "recursion" must be explained. Recursion is another concept explored in first-year computer science courses, and is a relatively complex concept. Formally, recursion is defined as "a programming method in which a routine calls itself."[5] It can also apply to a situation where a routine calls another routine which may call the first routine again, etc.

To understand recursion the analogy of a team of people handing a piece of paper around is helpful. One person begins to write on a particular paper, and hands it to another person when the need arises. That person writes on it and passes it again. The next person might pass it yet again, perhaps even back to the first person. Each person remembers who he got the paper from each time he received it. When one person is done working on the paper, he

4 Funk, Nathan. <http://www.singularsys.com/jep/doc/grammar.html>
5 <http://www.webopedia.com/TERM/r/recursion.html>

hands it back to the person who gave it to him. That person may or may not be done yet, and the paper might get passed around and handed back any number of times before it finally gets back to the very first person to work on it. When he is finished, the overall task is completed.

Recursion is used in the Paradox engine to build expression trees according to a grammar. There are separate routines for each level of the grammar. Each routine takes characters from the character string and builds its corresponding node and/or subnodes on the tree, passing off the string to a separate routine if necessary.

An example may help to illustrate this concept. Consider the following example expression along with a slightly modified version of the example grammar presented in the previous section:

$$7220 + \left(1 - \frac{13(2)}{5}\right)$$

```
Expression ::= Term { ( "+" | "-" ) Term }
Term       ::= { ( "×" | "/" ) Factor }
Factor     ::= Number |
               "(" Expression ")"
```

Using this grammar, three different routines would be written: Expression, Term and Factor. Each routine takes care of its respective line of the grammar. Note that the expression above could not be entered into the computer as shown, since the computer does not immediately understand graphics such as the large parentheses. Instead the expression must be entered in single-line form: `"7220+(1-13×2/5)"`

The recursion begins with the first line of the grammar and its corresponding routine: Expression. This routine first creates an Expression node as the root node of the expression tree on level 1. Then the routine follows the grammar's Expression specification to build the subnodes. The grammar specifies that an Expression consists of a Term first. Thus, the Term routine is called. It first creates a Term node as a subnode of the parent node (in this case, the Expression on level 1). Since a Term consists of a Factor first, the routine passes off the expression to Factor. In Factor, the routine checks for the presence of a digit or a left parenthesis. (Note that if it found anything else, an error would be reported and the parsing aborted.) It finds a digit, so it reads the entire number `"7220"` and places it into a Factor on level 3, a child node of the Term node on level 2. After that, Factor is done, so it passes control back to the Term routine that called it. The Term then looks for any remaining Factors preceded by a `"×"` or a `"/"`. The next character is a `"+"`, however, so its work is done and it passes control back to its parent, the root Expression node on level 1.

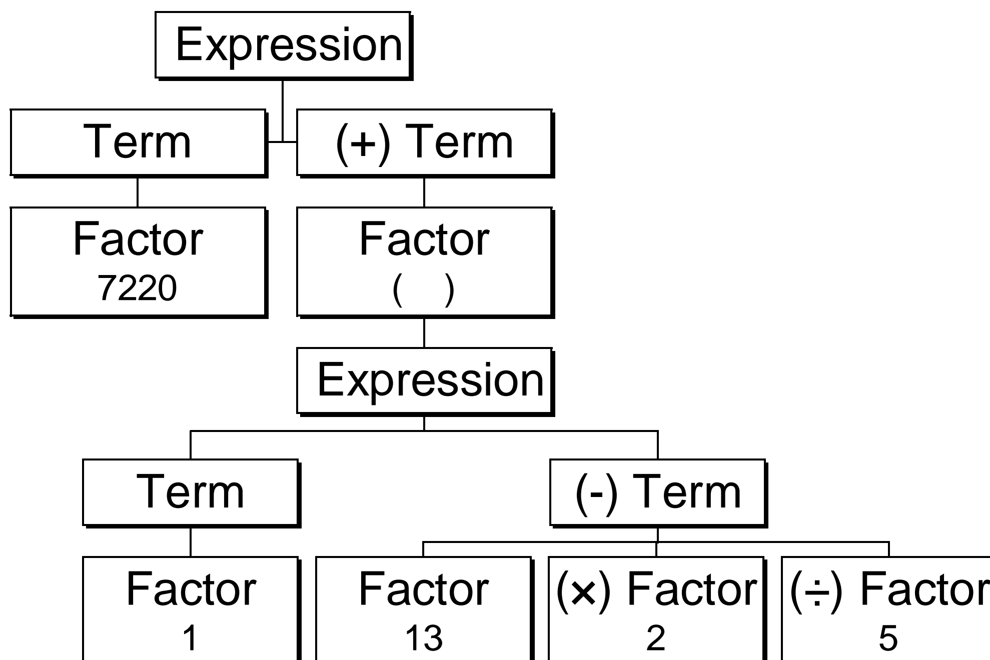
The Expression looks for any remaining Terms preceded by a `"+"` or a `"-"`. The next character is a `"+"`, so it calls the Term routine, passing the sign as a tag. The Term routine creates a Term node as another subnode on level 2 with a `"+"` tag before calling Factor. The Factor routine creates a Factor on level 3, then checks for the presence of a digit or a left parenthesis. It finds a left parenthesis, so it knows that it must read a subexpression. It reads

the left parenthesis and then calls the Expression routine. This is the purpose of recursion: that a routine far down in the hierarchy can call routines higher up, essentially starting the parsing over again.

So a new Expression is created on level 4. It then calls Term, which creates a Term on level 5. A factor is created, a number is read (since the next character is the digit "1"). Control passes back to the term on level 5. The next character is not a "×" or a "/", so control goes back again to the Expression on level 4. The next character is a "-" so Term is called, creating another Term on level 5 with the "-" tag.

Factor is called, creating a Factor with the number "13" on level 6. Back in Term, the next character is a "×", so Factor is called again. This time it creates a Factor on level 6 with a number ("2") and a "×" tag. Control passes back to Term, where the job is not done yet, because the next character is a "/". Factor is called again, creating yet another a Factor on level 6, this time with the number "5" and the "/" tag.

Control passes back to the Term on level 5. The next character is not a "×" or a "/", so control passes back once again to the Expression on level 4. The next character is not a " +" or a "-", however, so control goes back to its parent Factor on level 3. Remember that this routine is in the middle of reading a subexpression. It now checks for a right parenthesis and finds it. It reads the parenthesis and passes control back to its parent Term. The end of the expression has now been reached, and Term passes back to Expression. This is the root node, and since it is done (end of expression), the entire task is finished. Here is the completed expression tree:



This was a rather lengthy example. The irony is that this is a relatively short expression, parsed with a very simple grammar containing only three routines. Obviously, the human mind could very well handle this expression in much less time. However, it would take at least a minute or so, and would probably require the use of a calculator to crunch the numbers. On the computer, however, the affair takes on a whole new light. Although this algorithm might seem complicated, the computer does everything described above almost instantaneously, giving a result within a second or two. In addition, there is always the chance that a human might make a mistake in his arithmetic. Correctly programmed, however, the computer makes no mistakes. For these reasons, the relative complexity of the algorithm is more than made up for by the speed and reliability of the computer.

Implementation

This section deals with the actual Java implementation of the Paradox engine. Note that it does assume some experience with the Java language and with the idea of object-oriented programming. Note also that the source code is fairly well commented and following along in it while reading this section may be helpful in aiding understanding.

When I began Paradox, I knew little Java. Up until that time, I had mainly worked in Visual Basic and C++. When I began Paradox, therefore, I had to learn Java as I went. I am not an expert Java programmer now by any means, but I know quite a good deal more than when I began. Walter Savitch's book Java - An Introduction to Computer Science & Programming was extremely helpful in learning the Java language and the AWT. The section about Swing in The Java Tutorial[6] was also very helpful.

Nearly all of the code in Paradox was written by myself, and is formatted according to my own style. I copied certain snippets concerning certain topics like Swing (a GUI library like the AWT), but they were quickly edited and changed and are now probably quite unrecognizable. I believe it is fairly safe to say that the entire project is my own work.

The Paradox package contains 17 classes, each of which is described in detail in its respective section. Note that the only classes that the end developer should have to interact with are the ParadoxEngine, CalcIn, Function, Util, ParseException, EvalException and Expression classes. The other classes deal with the internal implementation of the engine and are declared private, thus prohibiting access to them by outside classes.

ParadoxEngine

This is the main class for the Paradox package. In fact, it might be the only class used by a particular program. For instance, the following function returns a double answer given an expression string:

```
public static double EvalExpression(String expression)
{
    return (new ParadoxEngine()).Evaluate(expression);
}
```

It contains the following major functions for top-level expression parsing and evaluating:

6 Campione, Mary, et al. <http://java.sun.com/docs/books/tutorial>

- `double Evaluate(String)` – parses an expression string and then evaluates the expression tree, returning a double and printing all errors to System.out
- `double EvaluateE(String)` – parses an expression string and then evaluates the expression tree, returning a double and throwing exceptions
- `Expression Parse(String)` – parses an expression string and returns the root Expression node, printing all errors to System.out
- `Expression ParseE(String)` – parses an expression string and returns the root Expression node, throwing exceptions
- `double Evaluate(Expression)` – evaluates an expression tree, returning a double and printing all errors to System.out
- `double EvaluateE(Expression)` – evaluates an expression tree, returning a double and throwing exceptions

Each of these functions simply creates a CalcIn stream (for those that do parsing) and then calls Eval on the expression tree (for those that do evaluation).

ParadoxEngine also contains functions for adding as well as printing lists of constants and functions (for more info on custom functions, see the section on the Function class). It also provides for printing an expression tree with the `toString(Expression)` and `toString(String)` functions.

When the developer creates a new instance of the ParadoxEngine class, the constructor creates a new random number generator plus new variable, constant and function maps. It then initializes the built-in constants and functions, which are described in Appendix B. To add a new constant in code, use the `AddConstant(String, double)` function. (see the section on the Function class for info on adding new functions).

Another interesting function contained in the ParadoxEngine is the `handleFunction(String, Vector)` function. This rather unique function handles function calls within expressions. Given the name of the function ("sin", "sqrt", "rand", etc.) and the arguments (in the form of a Vector of Expressions), it will calculate the results, returning a double. Note, however, that this function is for internal use and is not accessible from outside the package.

CalcIn

When I started the project, I quickly realized the need for a flexible stream class. Since I had some rather specific requirements, I decided to write my own. CalcIn has two constructors. The default constructor creates a stream by reading a line of input from System.in (using the `Util.readLine()` function). If you already have your expression in the form of a String, however, you can just use the `CalcIn(String)` constructor. This is probably the most common way to initialize a CalcIn stream. The other functions are used for manipulating the stream:

- `charAt(int)` - returns the character at a specified index

- `charsLeft()` - returns the number of characters left
- `dump()` - returns the entire stream, including any characters already read
- `EOS()` - returns true if the end of the stream has been reached, false if not
- `peek()` - returns the next character
- `peek(int)` - returns the next n characters
- `peekPastIdentifierW(int)` - returns the next n characters, skipping a identifier and any whitespace first
- `pos() - returns the current position in the stream`
- `readAll() - reads to end of streamskip()` - skips a character in the stream
- `readChar()` - reads a character
- `readDouble()` - reads a double (including scientific notation
- `readIdentifier()` - reads an identifier (function/variable name)
- `readString(int)` - reads a string of a specified length
- `skip(int)` - skips n characters
- `skipComma()` - skips any comma
- `skipW()` - skips a character and any whitespace that follows it
- `skipW(int)` - skips n characters and any whitespace that follows them
- `skipWhitespace()` - skips any whitespace

Note that the difference between peeking and reading is that peeking does not advance the current position whereas reading does. Once a character has been read, it cannot be read again. Peeking simply gives an quick look at the characters that are about to be read.

Function

To add a custom function, you must write a new class that extends the Function class, and then pass an instance of that class to the `addFunction(String, Function)` function in ParadoxEngine. An example best illustrates this idea. The following is the code for a a function that simply check for two arguments and then adds them together, returning the sum:

```
import java.util.*;
import paradox.*;

public class F_add extends Function
{
    public double Handle(Vector args, ParadoxEngine eng)
            throws paradox.EvalException
    {
        if (args.size() == 2)
            // return sum of arguments
            return
```

```
                    eng.Evaluate(((Expression)args.elementAt(0))) +
                    eng.Evaluate(((Expression)args.elementAt(1)));
              else
                  throw new paradox.EvalException("add() requires" +
                                                 "exactly two arguments.");
          }
      }
```

The above code would go in a separate file (F_add.java). To add the function to an instance of ParadoxEngine, the following line would be used:

```
engineInstance.addFunction("add", new F_add());
```

The first argument is the name of the function (this must be unique among function names) and the second argument is an instance of the F_add function.

Util

The Util class contains many small utility functions that are declared static so as to be accessible from any location. Here are a description of the functions in Util:

- `main(String[])` - This is a small program that was used during testing. It runs in the console and prompts for expressions, parsing and evaluating them until the user enters "q" to quit.
- `formatDouble(double)` - Formats a double value and makes it look nice for displaying it on the screen by stripping all trailing zeros and decimal points. Returns the formatted string. Note that this function converts the number to a Float before formatting it; this is to avoid some of the precision problems with doubles.
- `readLine()` - Returns one line of input from System.in, first discarding the endline character.
- `spaces(int)` - Returns a string containing a given number of spaces.
- `isZero(double)` - Checks to see if the double is essentially equal to zero (as determined by the AEP constant)
- `areEqual(double, double)` - Checks to see if the two doubles are essential equal (as determined by the AEP constant)
- `isAlpha(char)` - Returns true if the character is a letter ("a"-"z", "A"-"Z")
- `isNumeric(char)` - Returns true if the character is a digit ("0"-"9")
- `isAlphaNum(char)` - Returns true if the character is a letter or a digit
- `DEBUG(String)` - Prints the given error message to System.out if the SHOW_DEBUG_INFO flag is set.
- `DEBUG(String, int)` - Prints the given error message preceded by a given number of spaces to System.out if the SHOW_DEBUG_INFO flag is set.

The Util class also contains definitions of TRUE (1.0) and FALSE (0.0), along with the AEP (Approximately Equal Precision) constant and the accessor function `getAEPConstant()`, which is used to determine if two doubles are approximately equal.

This constant is defaulted to 1.0×10^{-14}, but can be changed if desired by using the `setAEPConstant(double)` function.

ParseException

The ParseException class is an exception class which prepends the text "Error while parsing expression" to any error message. Otherwise, it behaves as any other exception class.

Note that there is also a ParseException class in the java.text package. If you import both java.text and paradox and then attempt to use ParseException, you will need to indicate which ParseException class you want by using the full package name (i.e. `java.text.ParseException` or `paradox.ParseException`)

EvalException

The EvalException class is an exception class which prepends the text "Error while evaluating expression" to any error message. Otherwise, it behaves as any other exception class.

ExpressionTreeNode

This is the abstract class that all of the other classes in Paradox are based upon. It represents one node in an expression tree. Subnodes are contained in a vector appropriately named "subNodes," and a "tags" vector for storing subnode operator information if needed (whether the subnode is to be multiplied or divided, for instance). Two constructors initialize these vectors.

The definition of ExpressionTreeNode also contains two abstract functions: `Parse(CalcIn)` and `Eval()`. The former uses the given CalcIn to fill in information for the node and any subnodes, while the latter recursively evaluates any subnodes, performs any node-specific calculations and then returns a double. Since these functions are declared abstract, all classes inheriting from ExpressionTreeNode must implement them. Another function, `toString(int)`, returns a string representation of the node. Occasionally this function will be overloaded to accept an operator and/or tag parameter.

Expression

```
Expression ::= OrExpression |
               *Identifier* AssignOp Expression
AssignOp   ::= "=" | "#=" | "+=" | "-=" | "*=" | "•=" | "×=" |
               "/=" | "÷=" | "%=" | "^="
```

The top-level node in an expression tree is always an Expression, and this is the only node class that is declared public (it can be accessed from outside the package). Note that it also has a special constructor, allowing you to let the Expression know if it is a "main" Expression (whether it is the root node). An Expression contains either a single OrExpression or an assignment (an identifier followed by an assignment operator and another Expression). There are 11 possible assignment operators:

- `"="` - Assigns the subnode value to the variable.

- "#=" - Assigns the subnode value as a constant (ie. it cannot be changed after the first assignment).
- "+=" - Assigns the previous value of the variable added to the subnode value.
- "-=" - Assigns the previous value of the variable subtracted from the subnode value.
- "*=" - Assigns the previous value of the variable multiplied by the subnode value.
- "•=" - Assigns the previous value of the variable multiplied by the subnode value.
- "×=" - Assigns the previous value of the variable multiplied by the subnode value.
- "/=" - Assigns the previous value of the variable divided by the subnode value.
- "÷=" - Assigns the previous value of the variable divided by the subnode value.
- "%=" - Assigns the remainder obtained by dividing the previous value of the variable by the subnode value.
- "^=" - Assigns the previous value of the variable raised to the subnode value.

When `Parse()` is executed, it first checks for an empty expression. Assuming that the expression is not empty, it then checks for an identifier by checking for a letter in the input stream. If it finds an identifier, it checks the characters following it to see if there are any of the possible assignment operators. If it finds one, then it assumes that this is an assignment and sets the appropriate boolean member variable. It reads the identifier, stores the operator in the tag member variable and then creates a new Expression as a subnode. Note that it creates an Expression, not an OrExpression. This allows the user to nest assignments, such as "a = b = c = 1", in which case all variables mentioned are assigned the value "1". If there is no assignment, then it simply creates an OrExpression as a subnode.

When `Eval()` is executed, it checks the assignment member variable to see if this is an assignment Expression. If not, then it simply calls the `Eval()` function of its subnode OrExpression. If it is an assignment, then it performs the assignment by changing the value stored in the appropriate place in the engine's variable or constant map, first checking to make sure the user is not trying to reassign a constant. After evaluating, `Eval()` saves the answer in the "ans" variable (if desired and if this is a root node) regardless of whether or not this was an assignment. It then returns the result of the OrExpression or the value assigned (whichever is appropriate). Note that this means that an assignment such as "a = 3" will return the value ("3").

The `toString()` function simply prints "Expression" preceded by the given number of spaces and followed by the assignment operator if applicable.

OrExpression

```
OrExpression ::= AndExpression { OrOp AndExpression }
OrOp         ::= "|" | "||"
```

An OrExpression contains at least one AndExpression followed by any number of other AndExpressions preceded by a boolean OR operator. The OR operator is the first of the supported boolean operators (the operators dealing only with "true" or "false" values)

supported by Paradox. It is also the boolean operator with the lowest precedence (ie. it is evaluated last). Because it is the lowest precedence, it must be parsed first. OR returns true (1.0) if any one of its subnodes evaluates to true (1.0). Note that there are two operators:

- `"||"` This operator performs a "short-circuited" OR. This means that once it finds one subnode that is true, it automatically returns true without evaluating the other subnodes. This could possibly result in small speed gains at the expense of possible logic errors. For instance, `"1||x=3"` returns true since the first part ("1") is true. Since the short-circuited operator is used, the second part is not even evaluated, and thus x is not assigned the value 3 as was probably intended.
- `"|"` This is the "non-short-circuited" OR operator. This causes the engine to evaluate all subnodes before returning a value.

Note that if multiple operators are used in a single expression (`"1|1||1"`, for example), the last operator mentioned overrides all previous operators. Thus, the expression above is evaluated with the shortcircuited operator even though the non-shortcircuited operator is used earlier.

When `Parse()` is executed, it first creates a child AndExpression. It then checks for an OR operator. If it finds one, it first notes which one it found by setting the "shortcircuit" member variable flag and then creates child AndExpression. It does this repeatedly until it does not find another OR operator.

When `Eval()` is executed, it checks the shortcircuit member variable to see if this is a short-circuited OrExpression. If it is short-circuited, it starts evaluating the subnodes, returning true after the first true subnode (and returning false if none of the subnodes evaluate to true). If it is not short-circuited, it starts evaluating the subnodes, returning a value only after the last subnode has been evaluated.

The `toString()` function simply prints "OrExpression" preceded by the given number of spaces.

AndExpression

```
AndExpression ::= EqualExpression { AndOp EqualExpression }
AndOp         ::= "&" | "&&"
```

An AndExpression contains at least one EqualExpression followed by any number of other EqualExpressions preceded by a boolean AND operator. Unlike OR, AND returns true (1.0) if and only if ALL of its subnodes evaluate to true (1.0). Note that there are two operators:

- `"&&"` This operator performs a "short-circuited" AND. This means that once it finds one subnode that is false, it automatically returns false without evaluating the other subnodes. This has the same pros and cons as the short-circuited evaluation indicated by the corrosponding OR operator.
- `"&"` This is the "non-short-circuited" AND operator. This causes the engine to evaluate all subnodes before returning a value.

When `Parse()` is executed, it first creates a child EqualExpression. It then checks for an AND operator. If it finds one, it first notes which one it found by setting the "shortcircuit" member variable flag and then creates a child EqualExpression. It does this repeatedly until it does not find another AND operator.

When `Eval()` is executed, it checks the shortcircuit member variable to see if this is a short-circuited AndExpression. If it is short-circuited, it starts evaluating the subnodes, returning false after the first false subnode (and returning true if all of the subnodes evaluate to true). If it is not short-circuited, it starts evaluating the subnodes, returning a value only after the last subnode has been evaluated.

The `toString()` function simply prints "AndExpression" preceded by the given number of spaces.

EqualExpression

```
EqualExpression ::= RelationalExpression [ EqualOp RelationalExpression ]
EqualOp         ::= "==" | "!=" | "~="
```

An EqualExpression contains at least one RelationalExpression possibly followed by an operator and another RelationalExpression. This means that there can be at the most two child RelationalExpressions in an EqualExpression. There are three operators, all dealing with the equality or inequality of the EqualExpression's subnodes:

- `"=="` This operator compares the two subnodes, checking for strict equality on the lowest level. This is more appropriate for integers than doubles (which often have small errors in precision), and this is why the other `"~="` operator has been provided (see below). This operator only returns true (1.0) if the subnodes are equal. `"2==2"` returns true (1.0), while `"2==3"` returns false (0.0). Note that this operator is `"=="`, not `"="`. This is to prevent any ambiguity between it and the assignment operator. This trick is not unique to Paradox; it is done by many major programing languages, such as C++ and Java itself.
- `"!="` This operator is the reverse of the previous operator, and returns true (1.0) if the subnodes are NOT equal. `"2==2"` returns false (0.0), while `"2==3"` returns true (1.0).
- `"~="` This operator is used for comparing doubles, and only returns true if the difference between two numbers can be considered so small that the numbers are essentially equal. The precision of this check is determined by the AEPConstant member variable of the Util class.

The `Parse()` function is fairly straightforward. It creates a child RelationalExpression before checking for the presence of an equals operator. If it finds one, it saves the operator in the "op" member variable and then creates another child RelationalExpression.

The `Eval()` function is equally straightforward. It evaluates the first RelationalExpression, and then checks the "op" variable to see if there is an operation occurring. If so, it evaluates the second RelationalExpression, and then performs the operation and returns the result. Note that it uses the `Util.areEqual(double, double)` method to determine approximate equality ("~=").

The `toString()` function prints "EqualExpression" preceded by the given number of spaces and followed by an operator (if appropriate).

RelationalExpression

```
RelationalExpression ::= AdditiveExpression [ RelationalOp AdditiveExpression ]
RelationalOp         ::= "<" | ">" | "<=" | ">="
```

The RelationalExpression is the last of the main boolean operation classes. A RelationalExpression contains at least one AdditiveExpression possibly followed by an operator and another AdditiveExpression. This means that there can be at the most two child AdditiveExpressions in a RelationalExpression. There are four operators, all dealing with the comparison of order on the number line of the RelationalExpression's subnodes:

- `"<"` This operator compares the two subnodes, returning true if the first subnode is less than the second and false otherwise. `"2<3"` returns true (1.0), while `"2<1"` and `"2<2"` return false (0.0).
- `">"` This operator compares the two subnodes, returning true if the first subnode is greater than the second and false otherwise. `"3>2"` returns true (1.0), while `"3>4"` and `"3>3"` return false (0.0).
- `"<="` This operator compares the two subnodes, returning true if the first subnode is less than or equal to the second and false otherwise. `"2<=3"` and `"2<=2"` return true (1.0), while `"2<=1"` returns false (0.0).
- `">="` This operator compares the two subnodes, returning true if the first subnode is greater than or equal to the second and false otherwise. `"3>=2"` and `"3>=3"` return true (1.0), while `"3>=4"` returns false (0.0).

The `Parse()` function is fairly straightforward. It creates a child AdditiveExpression before checking for the presence of an relational operator. If it finds one, it saves the operator in the "op" member variable and then creates another child AdditiveExpression.

The `Eval()` function is equally straightforward. It evaluates the first AdditiveExpression, and then checks the "op" variable to see if there is an operation occurring. If so, it evaluates the second AdditiveExpression, and then performs the operation and returns the result.

The `toString()` function prints "RelationalExpression" preceded by the given number of spaces and followed by an operator (if appropriate).

AdditiveExpression

```
AdditiveExpression ::= MultiplicativeExpression { AddOp MultiplicativeExpression }
AddOp              ::= "+" | "-"
```

The AdditiveExpression is the first of the non-boolean mathematical operators (ie. they return values other than "true" or "false"). Addition and subtraction are together on the lowest precedence level, and they are both parsed in this class. An AdditiveExpression contains at least one MultiplicativeExpression followed by any number of other MultiplicativeExpressions preceded by an operator. There are only two operators: `"+"` representing addition, and `"-"` representing subtraction.

Note that this class is equivalent to the Term level used in the example grammar in the previous sections about theory.

`Parse()` is fairly simple. It creates one MultiplicativeExpression and then repeatedly looks for an operator followed by another MultiplicativeExpression until there are no more or until the stream is empty. It saves the operators in the tags vector.

`Eval()` is also fairly simple. It evaluates the first MultiplicativeExpression, and then it goes through the others, adding or subtracting their values from the first according to the operators contained in the tags vector.

The `toString()` function prints "AdditiveExpression" preceded by the given number of spaces and followed by an operator (if appropriate).

MultiplicativeExpression

```
MultiplicativeExpression ::= UnaryExpression { PowerExpression |
                                               MultOp UnaryExpression }
MultOp                   ::= "*" | "•" | "×" | "/" | "÷" | "%"
```

The MultiplicativeExpression is the second of the non-boolean mathematical operators. Multiplication and division are together on this level. A MultiplicativeExpression contains at least one UnaryExpression followed by any number of PowerExpressions or other UnaryExpressions preceded by an operator. There are several operators on this level: "`*`", "`•`" and "`×`" representing multiplication, "`/`" and "`÷`" representing division and "`%`" representing the modulus operator (which performs a division but returns the remainder).

Note that this class is equivalent to the Factor level used in the example grammar in the previous sections about theory.

The `Parse()` function first creates one child UnaryExpression. Then it goes into a loop where it adds other subnodes depending on what it finds. If it finds one of the operators, it saves the operator and creates a UnaryExpression. If it finds a number, it creates a PowerExpression, and if it finds a left parenthesis, it creates a UnaryExpression. These last two actions allow for implied multiplication. "`2 3`" and "`2(3)`" both evaluate to 6 because of this methods. Note that "`2 -3`" is interpreted as "2 minus 3" rather than "2 times -3." To use implied multiplication with a negative number, it must be inclosed in parentheses (ie. "`2(-3)`". All operators are saved in the tags vector.

Unlike `Parse()`, `Eval()` is fairly simple. It evaluates the first UnaryExpression, and then it goes through the others, multiplying or dividing their values from the first according to the operators contained in the tags vector. As mentioned before, the modulus operator returns the remainder of a division (ie. "`21%5`" returns 1, the remainder when 21 is divided by 5).

The `toString()` function prints "MultiplicativeExpression" preceded by the given number of spaces and an operator (provided by the parent AdditiveExpression).

UnaryExpression

```
UnaryExpression ::= UnaryOp UnaryExpression |
                    PowerExpression
```

```
UnaryOp          ::= "+" | "-" | "!" | "¬"
```

All operations to this point have been “binary” operators, meaning that they dealt with two operands. The UnaryExpression represents any possible “unary” operators (other than exponents), meaning that they deal with only one operand. A UnaryExpression contains either another UnaryExpression preceded by an operator or a PowerExpression. There are four possible unary operations:

- “+” - Positive. This operator is not really necessary (it is assumed: “ 2” is the same as “+2”), but is included for completeness.
- “-” - Negative. Note that these can be strung together: “--2” is equal to “2” since the negative signs cancel each other out.
- “!” - Boolean NOT. This returns true if the subnode is false, and false if it is true (ie. “!1” returns false (0) and “!0” returns true (1)).
- “¬” - Boolean NOT (see above).

The `Parse()` function is fairly simple. It first checks for an operator. If it finds one, it reads it, saves it in the tag member variable, and then creates a child UnaryExpression. Note that an UnaryExpression is created, not a PowerExpression. This allows the user to string together unary operations (ie. “--2” or “---3”). If no operator is found, it creates a child PowerExpression.

The `Eval()` function first checks the tag for an operator. If it finds one, it evalutes the child UnaryExpression and then applies the specified operator. If it does not find an operator, it just returns the value of the child PowerExpression.

The `toString()` function prints “UnaryExpression” with any sign or operator (provided by its parent MultiplicativeExpression).

PowerExpression

```
PowerExpression ::= UnaryExpressionNotPlusMinus [ PowerOp |
                                                  "^" UnaryExpression ]
PowerOp          ::= "¹" | "²" | "³"
```

The PowerExpression class takes care of any exponent that a number or expression might have. There are two types of exponents: single-character exponents (ie. “x^2”) and exponents expressed with the “^” operator (ie. “2^4”). Note that all single-character exponents can be expressed with the “^” operator (ie. “^{3}” equals “^3”). A PowerExpression is made up of a UnaryExpressionNotPlusMinus, which is the lowest level in the hierarchy, possibly followed by an exponent. Since there are two types of exponents, there are two possibilities for the exponent: a single exponent operator, or the power operator followed by the exponent (in the form of a UnaryExpression).

Note that there are three different single-character exponent operators: “^{1}”, “^{2}” and “^{3}”. The “^{1}” operator is probably unnecessary since it doesn't really do anything (any number raised to the first power is the same number: $x^1 = x$), but it is included for completeness.

The `Parse()` function first creates a child UnaryExpressionNotPlusMinus and then examines the next character. If it is a "^", it creates a UnaryExpression, storing it in the "exponent" member variable. If it is one of the exponential operators, it creates a UnaryExpression with the corresponding value and stores it in the exponent member variable.

The `Eval()` function simply evaluates the child UnaryExpressionNotPlusMinus and then checks for an exponent. If there is one, the proper calculations are performed and the result returned.

The `toString()` function prints "PowerExpression" proceded by any operator (provided by its parent MultiplicativeExpression) and followed by any exponent.

UnaryExpressionNotPlusMinus

```
UnaryExpressionNotPlusMinus ::= *Double* |
                                *Identifier* |
                                *Identifier* "(" ArgList ")" |
                                LParen Expression RParen
ArgList                     ::= [ Expression { "," Expression } ]
LParen                      ::= "(" | "[" | "{"
RParen                      ::= ")" | "]" | "}"
```

The UnaryExpressionNotPlusMinus is the lowest-level class. It represents a single number (double), variable (identifier), function call (identifier followed by "(", an argument list and ")") or subexpression (left parenthesis followed by an Expression and right parenthesis).

The `Parse()` function cues off of the first character:

- If it is a LParen ("(", "[" or "{"), then it assumes that there is a subexpression surrounded by parentheses. It reads the left parenthesis, creates a child Expression and then reads a RParen (")", "]" or "}"), throwing an exception if it does not find one. Note that all of the different parenthesis styles are treated the same (ie. "2•(3+4)" is no different from "2•[3+4]" or "2•{3+4}" or even "2•(3+4]"). The "tag" member variable is set to "e", denoting a subexpression.
- If it is a letter, it assumes that it is the first character of an identifier. It reads the identifier, saving it in the "identifier" member variable. It then peeks at the next character, checking to see if it is a left parenthesis ("("):
  - If it is a left parenthesis, it assumes that the identifier is the name of a function. It then goes into a loop to read the arguments, which are read as Expressions and stored as subnodes. Any errors are thrown as exceptions during this process. In addition, the tag is set to "f" to indicate a function call.
  - If there is no left parenthesis it is assumed that the identifier is a variable, and the tag is set to "v" to indicate this fact.
- If it is a number, it assumes that it is the first character of a double. It reads the double, storing it in the "value" member variable. It also sets the tag to "d" to denote a double.

The `Eval()` function cues off of the value stored in the "tag" member variable. If it is an "e", it evaluates the subexpression stored in the "subExpr" member variable and returns the

result. If it is a “f”, it sends the identifier and all subnodes to the `handleFunction()` producedure in the ParadoxEngine class. If it is “v”, it returns the variable stored in the constant or variable map for the specified identifier. Otherwise (if it is “d”), it simply returns the double stored in the “value” member variable.

The `toString()` function also cues off of the value stored in the “tag” member variable. If it is an “e”, it prints “UnaryExpressionNotPlusMinus (subexpression)” followed by the string representation of the subexpression. If it is a “f”, it prints “UnaryExpressionNotPlusMinus (function call)” followed by the function name and string representations of all subnodes. If it is “v”, it prints “UnaryExpressionNotPlusMinus (variable)” followed by the variable name. Otherwise (if it is “d”), it prints “UnaryExpressionNotPlusMinus” followed by the string representation of the double.

Calc4

Unless they're using Unix or one of its variants, it is fairly uncommon for regular computer users to utilize the command line much anymore. In this era of "user-friendly" interfaces, it is rare to find someone who would not prefer a "pretty" GUI (Graphical User Interface) to a "boring" CLI (Command Line Interface). Although there are those who would argue for the superiority of a CLI and there are others who would never dream of using anything other than a GUI, I believe that both have their place. Therefore, although Paradox has a simple CLI program built-in (found in the Util class), I have also created Calc4, a small sample GUI for demonstration purposes. It is by no means a comprehensive program, but it provides a reasonably nice interface for entering expressions and getting their results. It also demonstrates the possibility of using a simple scripting language in conjunction with Paradox to automate routine calculations.

Basic Interface

The main purpose of Calc4 is to provide a decent multiline display with which you can enter expressions and obtain results. When the program is started, this display is the dominating feature. To evaluate an expression, simply type the expression in the window and press the ENTER key (note that if for some reason you wish to simply insert a carriage return/line feed, you can do this by pressing Ctrl-Enter or Alt-Enter). This window does support copy and paste, so you don't have to enter complex expressions over and over again; simply copy and paste the expression and then edit it if necessary.

Also shown in the initial screen is a smaller window below the main display. This window serves as an error log, keeping track of all errors that occur in the session. If you enter an invalid expression, such as "3+*4", the error message "Error while parsing expression: Number expected but not found." is shown in this window. The error messages are fairly self-explanatory, but sometimes in a long expression it can be hard to tell exactly where the problem is. Therefore, it is recommended that you make extra effort to work slowly and enter your expressions correct the first time. However, typos are inevitable, and this error log helps alleviate some of the aggravation that they cause.

There is also a large set of buttons that can be shown by selecting "Show Buttons" from the View menu or by pressing F4. These buttons simply replace the current selection with the text

on them. When a button with text ending in parentheses (such as a function) is clicked, the cursor is moved inside the parentheses.

Also in the View menu is an option to show expression trees. If this option is selected, the expression tree is printed after the result when you enter an expression. The string representation used is obtained by calling the `toString()` function of the root Expression (see above section on the implementation for details).

The Engine menu contains options for displaying lists of supported constants and variables as well as for executing scripts (see next section for more info on scripts).

There are several other purely cosmetic options as well. You can change the font size and the "look & feel" of the application using the appropriate submenus in the View menu. In addition, selecting "Clear" in the menu or pressing Ctrl-Del will clear the display.

Scripts & Scripting

If you have some specific calculations that you perform over and over again, you can automate them with a script. A script is simply a text file that contains a series of expressions, commands and comments, with each on a separate line. A command is a line that begins with the "`@`" symbol. The next identifier after that symbol should be one of the following commands:

- `exec` - Treats the rest of the line as the filename of an external script to run.
- `if` - Evaluates the rest of the line as an expression, and then skips the following line if the result is false (0.0).
- `label` - Marks a place in code for use with a `@goto` statement. The next identifier on the line is used as the name of the label.
- `goto` - Moves the current script location to the specified location marked by a`@label` command or a "`:`" symbol.
- `print` - Evaluates the rest of the line as an expression and prints the results without an endline and without printing the expression first.
- `println` - Evaluates the rest of the line as an expression and prints the results with an endline but without printing the expression first.
- `printex` - Evaluates the rest of the line as an expression and prints the results with the expression first but without an endline.
- `printexln` - Evaluates the rest of the line as an expression and prints the results with the expression first and with an endline.
- `disp` - Treats the rest of the line as text to be printed onto the screen without an endline character.
- `displn` - Treats the rest of the line as text to be printed onto the screen with an endline character.
- `error` - Treats the rest of the line as text to be printed into the error log with an endline character.

- `abort` - Ends the script with an error message.
- `end` - Ends the script without an error message.
- `rem` - Ignores the rest of the line (comment).

A comment can also be indicated by placing "`//`" or "`'`" at the beginning of the line. Similarly, a label can also be indicated by placing a "`:`" at the beginning of the line. Any line that is not a command, comment or label is interpreted as an expression to be evaluated by the engine. These are normally variable assignments, although they could also be used to seed the random number generator with the `seed()` engine function.

The following is an example script for solving quadratic equations:

```
' QUADRATIC.TXT
'
' Calculates solutions to a quadratic equation.
'

@displn
@displn ---  Quadratic Solver  ---
@displn        Ax² + Bx + C
@disp A =
@println a
@disp B =
@println b
@disp C =
@println c

' calculate discriminant
d=b²-(4•a•c)

@if d<0
   @goto no_real
@if d==0
   @goto one_real
@if d>0
   @goto two_real
@end

: no_real
   '@displn No real solution.
   d=-d
   @disp Solution:
   @print (-b)/(2•a)
   @disp  ±
   @print sqrt(d)/(2•a)
   @displn i
@end

: one_real
   @disp Solution:
   @println (-b)/(2•a)
@end

: two_real
```

```
        @disp Solution:  (
        @print (-b-sqrt(d))/(2•a)
        @disp ,
        @print (-b+sqrt(d))/(2•a)
        @displn )
    @end
```

This script prints some opening info, calculates the discriminant, checks to see what kind of solution there is, branches to the appropriate location and then calculates and displays the results. Note that "a", "b" and "c" must be assigned before executing the script.

This is a very simple scripting language and is fairly limited, but it does demonstrate the exciting possibilities. Paradox has progressed to such a level that it can now itself be programmed!

Conclusion

Potential Applications

Paradox has many potential practical applications:

- It was designed first and foremost to be an easy-to-use library for parsing and evaluating infix mathematical expressions. Any programmer working on a project in Java can use Paradox to provide full expression calculation capabilities without having to "reinvent the wheel." For instance, someone writing a small expense manager could use Paradox to allow users to enter a mathematical expression such as "19.95(.045)" in the Amount field instead of having to use a calculator to obtain a single number first. Using Paradox, this addition would require only a few minutes on the part of the programmer, and would be a very practical addition to his program.
- Engineers who use computers to run calculations might also be able to use Paradox. They might wish to re-write parts of it for increased precision and speed, but the basic ideas would stay the same. Since they can then use Paradox to write their own programs, they can let the computer do the hard work while they concentrate on providing the best service possible to their customers or employers.
- Students in statistic, physics or other mathematics-based classes who need to be able to evaluate complicated expressions but who do not have a multi-line calculator could benefit from using Paradox in conjunction with Calc4. The addition of statistical functions to Paradox and graphing capabilities to Calc4 (see next section) would bring the project close to the basic capabilities of a TI-82 or TI-83 graphing calculator, which these students would normally be using. In addition, since Paradox is freely distributed, teachers could use it in classrooms and computer labs without worrying about copyrights or licensing restrictions.

Future Additions & Enhancements

Although I have put a lot of work into Paradox, it is by no means a completed project. There are many ways that Paradox could be improved:

- Support for complex numbers. This would probably require some major changes in the grammar, such as the addition of a ComplexExpression class. It would also require that all

Eval() functions be re-written to return a specialized number class (instead of a simple double) containing information about the imaginary part of the complex number. Many of the operations, such as multiplication, would also have to be re-written to handle complex numbers. The sqrt() function (and perhaps other functions) would have to be able to edited to return a complex number. These additions would be rather lengthy and would take quite a bit of time to implement and test.

- Support for number bases other than base-10, such as hexadecimal or binary. This would make the engine more useful for programmers and engineers.
- Support for multiple threads. The current implementation is not thread-safe (i.e. problems could arise if more than one thread tries to access a Paradox class at the same time). Implementing this support would require inserting "synchronized" into many of the function definitions and might require the rewriting of some of the functions, a process that could take some time.
- Statistical and/or calculus functions. The addition of certain statistical functions (such as standard deviation) would not be too difficult, while others (such as calculating binomial distributions or conducting statistical hypothesis tests) would probably require an external math library for the advanced computations.
- Graphing capabilities. Calc4 could be greatly enhanced by providing a mode that would allow users to graph functions (perhaps in a separate window). For instance, the user could enter the expression "`y=4 sin(x-3)`" and see the resulting sine wave. Multiple functions could be graphed on top of each other to compare them, and the display could be panned and zoomed to show different parts of the graph.
- The simple scripting language used by Calc4 could always be improved. Commands could be added to display prompts for numbers, for instance, rather than simply reading them from preassigned variables. While and for loops could be added as well as proper subroutines. In addition, a more sophisticated parsing routine (possibly recursive descent parsing) would probably have to be written once the language got too complex.
- Extensive testing. Since I have only two computers (both of them running Windows '98) and little time, I have not had a chance to really put Paradox or Calc4 through a comprehensive testing program. There are probably still some minor bugs and logic errors that may mess up certain kinds of calculations. No programmer can guarantee a perfect computer program. However, they can perform extensive testing to ascertain that their program is near-perfect. Unfortunately, I have not had time to do this kind of testing.

Distribution

I am currently distributing Paradox, Calc4 and their source code freely on the internet. They may be obtained at the following address:

```
http://www.freearrow.com/projects/paradox/
```

There is also an applet version of Paradox available at that website. Anyone with a browser capable of running Java applets can now use Paradox any time without having to download it (although downloading is still the preferred option, since calculations are slower when done by

the applet).

The Paradox engine is freeware. In other words, it is free to anyone who desires to use it in their application, whether it be for personal, educational or commercial usage. Any individual, organization or company who can afford to donate a small amount to compensate me for the time I spent on the project is certainly welcome to send any amount they deem appropriate, but payment is by no means required for the use of this engine. In addition, the Paradox engine may be freely distributed. If you wish to put it on a disk or in a code library, I would appreciate a complimentary copy of your disk or library.

The Java source code is also freely and easily obtainable for further study or enhancement. If you do wish to modify the engine and release the modified version, feel free to do so. You do not have to ask my permission, but it would be nice to notify me of your intentions beforehand. Note also that if you are planning to charge for your product, I would very much appreciate a complimentary copy of your product.

The only restriction on the use and distribution of this product is that you may not charge for an unmodified (or only marginally modified) version of this engine or claim that you wrote it.

All of the above also applies to Calc4, the user interface program

Closing

I have always liked programming, and I have especially enjoyed working on Paradox and Calc4. In fact, this has been one of the largest programming projects I have worked on, containing over 4,700 lines of code. It has been a long process, one with several setbacks and major problems, but the end result has been worth it. Paradox has more than met the expectations that I set out when I began, and I often use it myself.

I hope above all else that Paradox proves helpful to someone, whether they be a programmer, a student, an engineer or simply someone who needs the functionality that it provides. I hope that they enjoy using the program as much as I have enjoyed creating it.

Bibliography

Eck, David J. Introduction to Programming Using Java. Online textbook. <http://math.hws.edu/javanotes/index.html> Accessed: 3 Feb. 2003.

Funk, Nathan. “Java Mathematical Expression Parser.” <http://www.singularsys.com/jep/> Accessed: 3 Feb. 2003.

Mary Campione, et al. The Java Tutorial. Online book. <http://java.sun.com/docs/books/tutorial/> Accessed: 3 Feb. 2003.

Gosling, James, et al. Java Language Specification. Online book. <http://java.sun.com/docs/books/jls/second_edition/html/> Accessed: 5 Feb. 2003.

Litvin, Maria and Litvin, Gary. C++ for You++. AP Edition. Skylight Publishing, Andover, Massachusetts: 1998.

“Recursion.” Online dictionary. <http://www.webopedia.com/TERM/r/recursion.html> Accessed: 3 Feb. 2003.

“Recursive Descent Parsing” <http://cs.engr.uky.edu/~lewis/essays/compilers/rec-des.html> Accessed: 22 Jan. 2003.

Savitch, Walter. Java - An Introduction to Computer Science & Programming. Prentice Hall, Upper Saddle River, New Jersey: 1999.

“What is BNF Notation?” <http://cui.unige.ch/db-research/Engseignement/analyseinfo/AboutBNF.html> Accessed: 21 Jan. 2003.

Appendix A

PARADOX GRAMMAR

```
Expression ::= OrExpression |
               *identifier* AssignOp Expression
AssignOp   ::= "=" | "#=" | "+=" | "-=" | "*=" | "•=" | "×=" |
               "/=" | "÷=" | "%=" | "^="


OrExpression ::= AndExpression { OrOp AndExpression }
OrOp         ::= "|" | "||"


AndExpression ::= EqualExpression { AndOp EqualExpression }
AndOp         ::= "&" | "&&"


EqualExpression ::= RelationalExpression [ EqualOp RelationalExpression ]
EqualOp         ::= "==" | "!=" | "~="


RelationalExpession ::= AdditiveExpression [ RelationOp AdditiveExpression ]
RelationOp          ::= "<" | ">" | "<=" |">="


AdditiveExpression ::= MultiplicativeExpression { AddOp MultiplicativeExpression }
AddOp              ::= "+" | "-"


MultiplicativeExpression ::= [UnaryExpression] { PowerExpression |
                                                 MultOp UnaryExpression }
MultOp                   ::= "*" | "•" | "×" | "/" | "÷" | "%"


UnaryExpression ::= UnaryOp UnaryExpression |
                    PowerExpression
UnaryOp         ::= "+" | "-" | "!" | "¬"


PowerExpression ::= UnaryExpressionNotPlusMinus ( PowerOp |
                                                  "^" UnaryExpression )?
PowerOp         ::= "¹" | "²" | "³"


UnaryExpressionNotPlusMinus ::= *Double* |
                                *Identifier* |
                                *Identifier* "(" ArgList ")" |
                                LParen Expression RParen
ArgList                     ::= [ Expression { "," Expression } ]
LParen                      ::= "(" | "[" | "{"
RParen                      ::= ")" | "]" | "}"
```

Appendix B

CONSTANTS

Name	Value	Description
avogadro	6.022×10^{23}	atoms per mole of a substance
boltzmann	1.38×10^{-23}	
coulomb	8.99×10^{9}	
faraday	96,485	coulombs per mole of electrons
e	2.718281828459045	base of all natural logarithms
e_charge	-1.602×10^{-19}	an electron's charge
e_mass	9.11×10^{31}	an electron's mass
h_bar	1.05×10^{34}	$\hbar$
pi	3.141592653589793	
planck	6.626×10^{34}	Planck's constant
rydberg	2.18×10^{-18}	Rydberg's constant
speed_of_light	3×10^{8}	speed of light (m/s)

FUNCTIONS

Name	Description
abs(x)	absolute value of x - $\lvert x \rvert$
round(x)	x rounded to nearest whole number
ceil(x)	x raised to nearest whole number
floor(x)	x lowered to nearest whole number
int(x)	x lowered to nearest whole number
pow(x, y)	x raised to the yth power - x^{y}
inv(x)	inverse of x - x^{-1}
sqr(x)	x squared - x^{2}
sqrt(x)	square root of x - $\sqrt{x}$
sqrt(x, y)	yth root of x - $\sqrt[y]{x}$
exp(x)	e raised to the xth power - e^{x}
ln(x)	natural logarithm of x - $\log_{e} x$
log(x)	log base 10 of x - $\log_{10} x$
log(x, y)	log base y of x - $\log_{y} x$
log2(x)	log base 2 of x - $\log_{2} x$
log10(x)	log base 10 of x - $\log_{10} x$
sin(x)	sine of x
cos(x)	cosine of x
tan(x)	tangent of x

continued on next page...

```
sec(x)              secant of x
csc(x)              cosecant of x
cot(x)              cotangent of x
sind(x)             sine of x (in degrees)
cosd(x)             cosine of x (in degrees)
tand(x)             tangent of x (in degrees)
asin(x)             arcsine  -  sin-1 x
acos(x)             arccosine  -  cos-1 x
atan(x)             arctangent  -  tan-1 x
invsin(x)           inverse sine  -  sin-1 x
invcos(x)           inverse cosine  -  cos-1 x
invtan(x)           inverse tangent  -  tan-1 x
deg(x)              convert x radians to degrees
rad(x)              convert x degrees to radians
degrees(x)          convert x radians to degrees
radians(x)          convert x degrees to radians
min(x, y)           smaller value (x or y)
max(x, y)           larger value (x or y)
sum(x, ...)         sum of all arguments
avg(x, ...)         average of all arguments
seed(x)             seed the random number generator
time()              seconds since defined moment
rnd()               random number in range [0, 1)
rnd(x)              random number in range [0, x)
rnd(x, y)           random number in range [x, y)
rand()              random number in range [0, 1)
rand(x)             random number in range [0, x)
rand(x, y)          random number in range [x, y)
random()            random number in range [0, 1)
random(x)           random number in range [0, x)
random(x, y)        random number in range [x, y)
fact(x)             x factorial  -  x!
factorial(x)        x factorial  -  x!
```

Appendix C

JAVA GRAMMAR[7]

```
Identifier:
              IDENTIFIER
QualifiedIdentifier:
              Identifier { . Identifier }
Literal:
              IntegerLiteral
              FloatingPointLiteral
              CharacterLiteral
              StringLiteral
              BooleanLiteral
              NullLiteral
Expression:
              Expression1 [AssignmentOperator Expression1]]
AssignmentOperator:
              =
              +=
              -=
              *=
              /=
              &=
              |=
              ^=
              %=
              <<=
              >>=
              >>>=
Type:
              Identifier {   .   Identifier } BracketsOpt
              BasicType
StatementExpression:
              Expression
ConstantExpression:
              Expression
Expression1:
              Expression2 [Expression1Rest]
Expression1Rest:
              [  ?   Expression   :   Expression1]
Expression2 :
              Expression3 [Expression2Rest]
Expression2Rest:
              {Infixop Expression3}
              Expression3 instanceof Type
Infixop:
              ||
              &&
              |
              ^
              &
              ==
              !=
              <
              >
              <=
              >=
              <<
              >>
              >>>
              +
              -
              *
              /
              %
```

7 Gosling, James, et al. <http://java.sun.com/docs/books/jls/second_edition/html/syntax.doc.html>

```
Expression3:
              PrefixOp Expression3
              (   Expr | Type   )   Expression3
              Primary {Selector} {PostfixOp}
Primary:
              ( Expression )
              this [Arguments]
              super SuperSuffix
              Literal
              new Creator
              Identifier { . Identifier }[ IdentifierSuffix]
              BasicType BracketsOpt .class
              void.class
IdentifierSuffix:
              [ ( ] BracketsOpt   .   class | Expression ])
              Arguments   .   ( class | this | super Arguments | new InnerCreator )
PrefixOp:
              ++
              --
              !
              ~
              +
              -
PostfixOp:
              ++
              --
Selector:
              . Identifier [Arguments]
              . this
              . super SuperSuffix
              . new InnerCreator
              [ Expression ]
SuperSuffix:
              Arguments
              . Identifier [Arguments]
BasicType:
              byte
              short
              char
              int
              long
              float
              double
              boolean
ArgumentsOpt:
              [ Arguments ]
Arguments:
              ( [Expression { , Expression }] )
BracketsOpt:
              {[]}
Creator:
              QualifiedIdentifier ( ArrayCreatorRest  | ClassCreatorRest )
InnerCreator:
              Identifier ClassCreatorRest
ArrayCreatorRest:
              [ ( ] BracketsOpt ArrayInitializer | Expression ] {[ Expression ]} BracketsOpt )
ClassCreatorRest:
              Arguments [ClassBody]
ArrayInitializer:
              { [VariableInitializer {, VariableInitializer} [,]] }
VariableInitializer:
              ArrayInitializer
              Expression
ParExpression:
              ( Expression )
Block:
              { BlockStatements }
BlockStatements:
              { BlockStatement }
```

```
BlockStatement :
             LocalVariableDeclarationStatement
             ClassOrInterfaceDeclaration
             [Identifier :] Statement
LocalVariableDeclarationStatement:
             [final] Type VariableDeclarators   ;
Statement:
             Block
             if ParExpression Statement [else Statement]
             for ( ForInitOpt   ;   [Expression]   ;   ForUpdateOpt )         Statement
             while ParExpression Statement
             do Statement while ParExpression   ;
             try Block ( Catches | [Catches] finally Block )
             switch ParExpression { SwitchBlockStatementGroups }
             synchronized ParExpression Block
             return [Expression] ;
             throw Expression   ;
             break [Identifier]
             continue [Identifier]
             ;
             ExpressionStatement
             Identifier   :   Statement
Catches:
             CatchClause {CatchClause}
CatchClause:
             catch ( FormalParameter ) Block
SwitchBlockStatementGroups:
             { SwitchBlockStatementGroup }
SwitchBlockStatementGroup:
             SwitchLabel BlockStatements
SwitchLabel:
             case ConstantExpression   :
             default:
MoreStatementExpressions:
             { , StatementExpression }
ForInit:
             StatementExpression MoreStatementExpressions
             [final] Type VariableDeclarators
ForUpdate:
             StatementExpression MoreStatementExpressions
ModifiersOpt:
             { Modifier }
Modifier:
             public
             protected
             private
             static
             abstract
             final
             native
             synchronized
             transient
             volatile
             strictfp
VariableDeclarators:
             VariableDeclarator { ,   VariableDeclarator }
VariableDeclaratorsRest:
             VariableDeclaratorRest { ,   VariableDeclarator }
ConstantDeclaratorsRest:
             ConstantDeclaratorRest { ,   ConstantDeclarator }
VariableDeclarator:
             Identifier VariableDeclaratorRest
ConstantDeclarator:
             Identifier ConstantDeclaratorRest
VariableDeclaratorRest:
             BracketsOpt [  =   VariableInitializer]
ConstantDeclaratorRest:
             BracketsOpt   =   VariableInitializer
VariableDeclaratorId:
```

```
              Identifier BracketsOpt


CompilationUnit:
              [package QualifiedIdentifier   ;  ] {ImportDeclaration}
              {TypeDeclaration}
ImportDeclaration:
              import Identifier {   .   Identifier } [   .     *   ] ;
TypeDeclaration:
              ClassOrInterfaceDeclaration
              ;
ClassOrInterfaceDeclaration:
              ModifiersOpt (ClassDeclaration | InterfaceDeclaration)
ClassDeclaration:
              class Identifier [extends Type] [implements TypeList] ClassBody
InterfaceDeclaration:
              interface Identifier [extends TypeList] InterfaceBody
TypeList:
              Type {   ,   Type}
ClassBody:
              { {ClassBodyDeclaration} }
InterfaceBody:
              { {InterfaceBodyDeclaration} }
ClassBodyDeclaration:
              ;
              [static] Block
              ModifiersOpt MemberDecl
MemberDecl:
              MethodOrFieldDecl
              void Identifier MethodDeclaratorRest
              Identifier ConstructorDeclaratorRest
              ClassOrInterfaceDeclaration
MethodOrFieldDecl:
              Type Identifier MethodOrFieldRest
MethodOrFieldRest:
              VariableDeclaratorRest
              MethodDeclaratorRest
InterfaceBodyDeclaration:
              ;
              ModifiersOpt InterfaceMemberDecl
InterfaceMemberDecl:
              InterfaceMethodOrFieldDecl
              void Identifier VoidInterfaceMethodDeclaratorRest
              ClassOrInterfaceDeclaration
InterfaceMethodOrFieldDecl:
              Type Identifier InterfaceMethodOrFieldRest
InterfaceMethodOrFieldRest:
              ConstantDeclaratorsRest ;
              InterfaceMethodDeclaratorRest
MethodDeclaratorRest:
              FormalParameters BracketsOpt [throws QualifiedIdentifierList] (
              MethodBody |   ;  )
VoidMethodDeclaratorRest:
              FormalParameters [throws QualifiedIdentifierList] ( MethodBody   |   ;  )
InterfaceMethodDeclaratorRest:
              FormalParameters BracketsOpt [throws QualifiedIdentifierList]   ;
VoidInterfaceMethodDeclaratorRest:
              FormalParameters [throws QualifiedIdentifierList]   ;
ConstructorDeclaratorRest:
              FormalParameters [throws QualifiedIdentifierList] MethodBody
QualifiedIdentifierList:
              QualifiedIdentifier {   ,   QualifiedIdentifier}
FormalParameters:
              ( [FormalParameter { , FormalParameter}] )
FormalParameter:
              [final] Type VariableDeclaratorId
MethodBody:
              Block
```