Final Report
Polynomial Manipulation Language (PML)

COMS 4115 Programming Languages and Translators

Melinda Agyekum Shezan Baig Hari Kurup Subadhra Sridharan

December 18, 2003

Contents

Chapter 1
Introduction

Polynomial equations are mathematical representations of real world problems and are used in a variety of professional fields. These mathematical expressions are written as the sum of the products of numbers and variables. A few practical applications which rely heavily on polynomial expressions are: missile trajectory, weather forecasting, spacecraft re-entry, building construction, and financial market calculations.

One example of a real world problem where polynomial functions are applied is testing the effectiveness of a new drug. The quantity of medication given to a person under testing can be varied and the improvement or degradation can be noted. There may be other variables, which can be applied to the same problem, like the patient's age, weight, and other existing medical conditions, if any.

Regardless of the situation, the use of polynomials can provide an individual better insight into a problem. Although polynomials are extremely important components of algebra, solving these problems manually can be a time-consuming and tedious process. It is because of this, there is a need for a system, which will perform computations in a methodical way irrespective of the problem at hand.

The polynomial manipulation language (PML) tool is a programming language built for flexible manipulation of polynomial expressions. With its extensive set of built in operations and functions, PML can be used to specify an algorithm involving polynomials. In addition, this language is easy to understand, allowing this to be a user-friendly language for programmers to enjoy.

PML is designed for manipulating symbolic mathematical computations. In contrast to numerical computation, PML emphasizes computing with symbols representing mathematical concepts. The input to algorithms will be expressions or polynomial equations, while the output of the translations will be returned in algebraic form. From such an expression, one can deduce how the change in parameters will affect the result of computation. Although PML will be able to handle numbers and symbols with equal capacity, the primary role of this application is to facilitate symbolic computational programs.

1.1 Background

Given below is a brief description about types of polynomials and equations supported by PML.

Polynomials

A polynomial is a mathematical expression involving a sum of powers multiplied by coefficients. Broadly classified, there are two types of polynomials depending on

the number of unique variables within the equations. These types are called univariate and multivariate polynomials.

Univariate Polynomial

A polynomial expressed in one variable is known as a univariate polynomial. An example of a univariate polynomial can be found below in Eq. 1.

$$c_i x^i + c_{i-1} x^{(i-1)} + \ldots c_0 \qquad \textit{(Eq. 1)}$$

In the aforementioned expression c_i, c_{i-1},… terms each represent coefficients, while the superscripts represent the degree of each term. This polynomial has only one variable, which is represented by 'x'.

Multivariate Polynomial

A polynomial expressed in more than one variable is known as a multivariate polynomial. An example of a multivariate polynomial can be found below in Equation 2.

$$c_i x^i y^i + c_{i-1} x^{(i-1)} y^{(i-1)} + \ldots c_i \qquad \textit{(Eq. 2)}$$

This equation is expressed using two variables 'x' and 'y'.

Linear Equations

Polynomials equations of the form

$$ax + b = c \qquad \textit{(Eq. 3)}$$

are called linear equations, having only one variable whose degree is one. All other equations not in the form mentioned by Eq. 3 are non-liner equations.

Quadratic Equations

All polynomials equations of the form

$$ax^2 + bx + c = 0 \qquad \textit{(Eq. 4)}$$

are called quadratic equations. Quadratic equations are second order degree equations and the roots of the equation can be determined using the quadratic formula.

Roots and Factoring

A root of a polynomial *P(z)* is a number z_i such that $P(z_i) = 0$. A polynomial of n degrees has n roots.

A factor is any number that divides a given number evenly (without a remainder). If r is a root of a polynomial equation *f(x) = 0*, then *(x-r)* is a factor of the polynomial *f(x)*.

PML supports all of the above-mentioned types of polynomials and equations, and more.

1.2 LANGUAGE FEATURES

PML contains several features, enabling it to be viewed as one of the most powerful symbolic equation language tools. Brief descriptions of these features are listed below:

Symbolic Interpretation
Unlike several applications on the market, PML provides a symbolic representation of polynomial equations. This symbolism allows the user to gain a more theoretical perspective of the function being performed, maintaining the likeness of which most polynomials are represented in algebra.

Most of the existing languages do not have the capability to accept a polynomial in its natural form and then manipulate it. This deficiency implies that users must create self-devised methods to enter polynomials to the program. Below is an example of feeding programs into a language other than PML:

(*User Enters)*> 2 4 -3 3 5 0 (*Program Interprets)*> $2x^4 -3x^3 + 5x$

This input does not represent the actual polynomial representation. There can be many more of such creative input sequence. With PML the user will be able to directly enter the equation listed below.

$$2x^4 -3x^3 + 5x \qquad \textbf{\textit{(Eq. 5)}}$$

Language Commands
Functions and keywords provided by PML are similar to the standard mathematics terminology; this makes PML easy and intuitive to use. Individuals with a working knowledge of symbolic mathematics and some programming background can easily start coding useful programs in PML.

Function Performance
Through the use of a very intuitive language, users will be able to perform operations such as addition, subtraction, multiplication, division, factorization, simplification, and differentiation of polynomials.

Equation Evaluation
PML will be equipped to handle a few numerical evaluations. The language will have the capability to solve for the numerical roots of an expression. Also, expressions will be able to be evaluated, provided that the user enters a number, which will be substituted for a variable.

User Customization
In addition to built-in functions, users will also be able to enter a polynomial and provide the program with steps on how to manipulate the equation.

Elimination of Error
PML does not have pointers. As a result it is not possible to write programs that can corrupt memories and cause systems to crash, making PML a stable language.

1.3 LANGUAGE IMPLEMENTATION

Functional Language
As opposed to an object-oriented language, this language will be implemented as a traditional functional language. One primary reason for this decision is that a functional language is much easier for the user to understand. Also, object-oriented concepts are not necessary in our domain.

The user can invoke operations through function calls as well as create their own functions. Recursive functions are also supported in the language. Many standard functions addition, subtraction, multiplication, division, and differentiation will be stored in standard libraries. However, the users will not be restricted to use the standard functions, and they can choose to write their own functions to perform the above-mentioned operations. Users will also be permitted to create their own libraries that can be linked together with many other programs.

Interpreted
The lexical scanner and parser will be created using ANTLR. The ANTLR system will produce the necessary information for the PML interpreter to execute the user's program. One convenience of an interpreted language is that it does not have to be compiled into machine code. This enables the language to be ported to many different platforms and architectures. The users can run a PML interpreter that was designed for a specific platform without the need to recompile their existing source code. The interpreter will then interpret the source code on the fly. A user can also share source code among a group of peers and be rest-assured that there will not be any problems running it. This is because the PML interpreter will work the same on all platforms and environments.

1.4 PML SCOPE AND LIMITATION

PML is a preliminary venture in providing a language specifically built for symbolic

mathematics. As such, the introductory version of PML is not aimed at handling the entire gamut of polynomial operations. PML will handle addition, subtraction, multiplication, division, factorization and differentiation of polynomials. No features or language support will be provided for higher operations like partial differentiation or integration.

Chapter 2
Language Tutorial

As introduced earlier, a polynomial is a sum of terms and each term comprises of a coefficient, one or more variables raised to a degree. If you are familiar with simple programming constructs and have worked with simple polynomials before this tutorial will help you build programs that manipulate polynomials using PML in a short time.

A simple PML program consists of the global variable declaration part, function definitions, and a main function with optional variable declaration. In each of the functions, these variables are visible only to those functions under which they are declared. Each of these parts are enclosed within a 'begin..end' block.

2.1 A Simple PML Program

```
# add.pml
# author: XXXX YYYY
# This program adds and prints two polys and two terms

vars
  term t1 = 5X^6;
  term t2 = 5.9;
end

func void main()
begin
    vars
        poly p1 = 5X^2 + 7;
        poly p2 = 5X^3 + 8X^2;
    end
     addandprintpoly(p1, p2);
     print "Adding terms: ";
     print t1 + t2;
     print t2 +  t1;
end

func void addandprintpoly(poly p1, poly p2)
begin
    print "Adding Poly: ";
    print p1 + p2;
    print p2 + p1;
end
```

This simple program illustrates the different parts of a pml program. The main function

will be executed first when this program is compiled and run. The variables t1 and t2 are global variables as they are part of the global variable declaration. The variables p1 and p2 are declared in the main function and its visibility is restricted to the main function. The addandprintpoly(poly p1, poly p2) is a function that will accept two polynomials as arguments and add and print the results. Since p1 and p2 are local to main they are passed as arguments to the addandprintpoly function. A one line comment begins with #.

2.1 Compiling and Running a .pml program

Now that we have a source code, the next step is to get the code, compiled and running. In PML the only command we have to issue is this.

> java PML add.pml

where add.pml is the name of the source code seen in the example above. This command first compiles the file and then executes it, if the compilation process is successful.

NOTE: In order for this command to work properly you need to have your java environment and classpath set up properly and have all the .class files of pml in the same directory as your source code. You must also execute this command from the directory of the source code.

2.2 A more complex PML program

Now that we have seen how to write, compile, and run a simple PML program, we can move on and see a more complex example with more features of PML like conversion of polys to termarrays, arrays, use of some inbuilt functions and loops.

```
# This program differentiates a polynomial and prints the output

func poly differentiate(poly p)
begin
   vars
      int i;
      termarray ta;
      poly ret;
      term t = 2;
   end

   ta = polyterm(p);
   i = 1;

   while (i <= length(ta))
   begin
      ta[i] = (coeff(ta[i])*degree(ta[i]))X^(degree(ta[i])-1);
      ret = ret + ta[i];
```

```
        i = i + 1;
    end

    return ret;
end

func void main()
begin
    vars
        poly p = 3X^2 + 4 + 2X^4 + 9X;
    end

    print "Differentiation Test";
    print "~~~~~~~~~~~~~~~~~~~~";
    print "Original Poly:  ", p;
    print "Result Poly:  ", differentiate(p);
    print "PolyTerm: ", polyterm(differentiate(p));
end
```

The function differentiate(poly p) accepts a poly as an argument and differentiates it with respect to X and returns a poly. The datatype termarray stores the terms of the poly p as an array of terms. The while loops checks for the length of the termarray (which is equal to the number of terms in the poly p) and for each element in the termarray differentiates it and adds it to the poly called ret. When all the terms have been differentiated the poly ret is returned. The main function prints out the original polynomial and its differentiation.

This program uses four inbuilt functions called *length() , coeff(), degree(),* and *polyterm().*

> ***length()-*** accepts a termarray as the data type and returns the length of the termarray.
>
> ***polyterm()-*** accepts a poly as the input and returns a termarray with each of the terms of the poly as an element in the termarray. This helps in easier manipulation of the terms of a polynomial.
>
> ***coeff()-*** accepts a term as an argument and returns the coefficient of the term.
>
> ***degree()-*** accepts a term as an argument and returns the degree of the term. In a univariate it is just the degree of the only variable present. In the case of a multivariate polynomial it is the sum of the degrees of all the variables present in the term.

This program is a very good example of the power of PML in manipulating polynomials. A complicated operation like differentiation can be written in in just 35 lines of code.

Chapter 3
Language Reference Manual

PML as the name suggests is a polynomial manipulation language for symbolic mathematics. Each program written in PML is case-sensitive and can be written in standard ASCII file format. The grammar has been generated using the tool ANTLR.

3.1 LEXICAL CONVENTIONS

The tokens of PML are identifiers, keywords, and expression operators. All forms of whitespace (blanks, tabs, and newlines) and comments are ignored. Whitespace is used to separate identifiers.

For token parsing, the language uses a "greedy" approach, meaning that a token is compared to the longest possible matching character stream.

3.1.1 Comments

Both single and multi-line comments will be accepted, single using '#' and mulit-line comments using "#{" as the opening declaration. Example of multi-line and single line comments are below.

```
# This is an example of a single line comment
#{ This is an example of a multiple line comment
   because it covers more than one line }#
#{ This is also an example of a multiple line comment }#
```

3.1.2 Identifiers (Names)

An identifier is considered as a sequence of at least one letter followed by any number of letter, digits, or underscores. Identifiers must consist of lower case letters only.

Acceptable identifiers: abc, a1234, a_ldsa, b__
Unacceptable identifiers: 3213, 3_a, _232_, 1, A, aBC

3.1.3 Keywords

The following identifiers are reserved words and should not be used otherwise:

begin	end	poly	return
break	float	polyeq	vars
char	func	print	void
do	if	term	chararray
else	int	termarray	

3.1.4 Type Specifiers

Data types must be specified as one of the following types: int, float, term, poly, polyeq, termarray, char, and string literals. The language does not support the user created data types.

3.1.4.1 int – An optionally signed sequence of digit is an integer constant. These constants can hold the range of *-2,147,483,647 to +2,147,483,648*

int: 2, 233, -543, 01, +10, 99

3.1.4.2 float - A floating point consists of an integer part, a decimal point, and a fraction part. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the decimal point with a fraction part (not both) may be missing. All exponentials must be declared in decimal format.

float: 2, .34, .33, -4.02,

3.1.4.3 term – A term is an int or float followed by an optional number of variable and a power parts. These variables must be in upper case. A power part can only exist if a variable is present. Each capital letter in a term represents a variable. In a term there is an implicit '*' sign to indication multiplication between a variable and an int, float, or another variable.

Term: *2, 3X, 3XY^2, 2.3YZ, 42X^1Y^1*
term: *XY (has variables X and Y which are multiplied together)*

3.1.4.4 poly – A poly is considered as two or more terms separated with an addition operator. An int and a float can also be considered polys. A complete listing of the addition operators can be found in section 2.5.1

poly: 2+3X, 2, 34.034XY^2 + X^2, .3XY – 3Y, 4

3.1.4.5 polyeq – A polyeq consists of two or more polys followed by a comparison operator. Complete listing of relational and equality operators can be found in sections 2.5.3 and 2.5.4, respectively.

Polyeq: 4 = 3X2, 4X<2X+3Y^6, X+Y = X +1

3.1.4.6 termarray – An array of terms can be represented by the data type termarray. Individual items of the array can be accessed by the notation *termarray_variable* [*index number*]. The index number, which starts from 1, refers to the order in which items are stored in the array. The length of this array is dynamically allocated and it can be increased or decreased by the '+' and '-' operations. The length of the array is equal to the number of items in it and can be

obtained by using the in-built function *length(..)*, which is explained in a later section.

Termarray: t[1] → returns the first element in the array.

3.1.4.7 char An object of type char can be used to store any member of length one, belonging to the ASCII character set.

Char c = 'x' ; the variable c will now have the value of 'x'.

3.1.4.8 chararray - An array of characters which has the same properties of this that of termarray *(See 2.4.6)*.

3.1.4.9 string literals A string literal also called a string constant, is a sequence of characters surrounded by double quotes, as in "…". String literals, can only be used with the print statement.

3.2 CONVERSIONS

Implicit type conversions will be supported for the following:

int → float
float → int (only if no fraction part)
float → term
term → poly
char → term

The following explicit conversions are available:

1. (poly -> termarray) which is done explicitly using the polyterm() function
2. (poly -> chararray) which is done explicitly using the variable() function

The following typecasts are available:

1. float -> int (fraction part discarded)
2. term -> float (only if coefficient can be converted to float)
3. poly -> term (only if poly has 1 term)

3. -> float is converted using the coeff(…) method.

3.3. EXPRESSIONS

3.3.1 Identifiers

An identifier is a primary expression provided it has been declared as explained below. Its type is specified in the declaration.

3.3.2 *(expression)*

A parenthesized expression is identical to an expression without parenthesis.

3.3.3 Operators

Operators are used to do polynomial and term manipulation. The types of operators supported are additive, multiplicative, relational, equality, and power.

3.3.3.1 Multiplicative – '*', '/' are multiplicative operators and used to perform multiplication and division between polynomials and terms. These operators have a higher precedence than additive operators.

3.3.3.1.1 *expr* * *expr* is an expression implying multiplication. If both operands are *int* then the resulting expression is an *int*. If both operands are *float* then the resulting expression is a *float*. If one operand is a *float* and the other operand is an *int* then the resulting expression is of type *float*.

Float * int → 3.4 * 5,
int * int → 10 * 20,
float * float → 2.5 * 7.6

Multiplicative operators applied to any other data type except int and float will result in an error

3.3.3.1.2 *expr / expr* is an expression implying division. Multiplication conversion rules from section 3.4.3.1 apply.

Float / int → 3.4 / 5,
int / int → 10 / 20,
float / float → 2.5 / 7.6

3.3.3.2 Additive – '+' and '-'are additive operators which group from left to right. These operators will be used in between terms as well as to add and subtract two polynomials. These terms are also used to denote positive and negative values. If the '+' is not explicitly implied values are assumed positive.

3.3.3.2.1 *expr* + *expr* is an additive expressions and the result is also an expression. The '+' operator is used for addition of all variables. For integers and floats '+' performs numerical addition. With variables such as term, char, termarray and poly, the '+' is used as a binary addition operator.

When the operands are like terms with same degree the operator returns a single value whose coefficient is the sum of the coefficients of the operands and the degree is the same as that of the operands. In binary addition, the operands with unlike terms return a polynomial which is the concatenation of the two terms. The magnitudes of the coefficients of the operands are maintained in the returned polynomial.

$$2X+3X \rightarrow 5X$$

When the operands are a term and a polynomial or a polynomial and a polynomial of different variables and degrees, the return value is a polynomial, a concatenation of the two operands. The magnitudes of the coefficients of the operands are maintained in the returned polynomial.

3X^2 + (4X +Y+Z) → 3X^2 + 4X +Y +Z
(2XY+Z) + (Y+Z) → 2XY +Z +Y+Z

If an integer or float is being added to a term or polynomial the result is a single polynomial, which is the concatenation of the operands. The integer/float is treated as a term with zero variables and degree and the resulting polynomial maintains the magnitude of the operands.

3 + (3XY^2) → 3+3XY^2

When the operands are characters of same value, the result is a single value returned as a polynomial. The characters are considered as terms with a coefficient and degree of one. The result is the sum of the two terms.

X + X → 2X

When the operands are non-similar characters, the result is a polynomial which is the concatenation of the two characters.

X + Y → X + Y

Termarray operands added to any non-termarray (int, floats, or term) operands result in a termarray whose length increases by one and the new element in the array is the non-termarray parameter. If the operands are a termarray and a polynomial, '+' will break poly into its constituent terms and append these terms to termarray. For example:

```
termarray ta;
poly p = 2X^2 + 3X + 4;
```

```
ta = ta + p;
```

result is a termarray that has 2X^2, 3X and 4 as its three elements.

3.3.3.2.2 *expr – expr* is a subtraction expression and the result is an expression. The type of the expression is determined by the type definition in section 4.3.2.1, except the '-'is used to return the difference of coefficients.

Poly op term → *(2X+4Y) – 2YZ*

Another distinction between addition and subtraction is the distribution of a negative sign through a term. If the object on the right-hand side of the subtraction sign is a polynomial, the minus is then distributed through to all the terms of the polynomial, changing the magnitude of the terms (i.e. '+' to '-'and '-'to '+'). The left hand side is then concatenated with the right hand side to form a polynomial.

2X – (4Y +YZ –Z^2) → 2X – 4Y – YZ + Z^2

The change in magnitude is partially due to the internal representation of terms in the system. Internally the parenthesis is not maintained and as a result for a '-' operation to store the proper value of every term, change in magnitude is necessary.

When the operands are termarray and an int, float, character, or term, the minuend has to be of type termarray. In such a case the non-termarray operand is removed from the termarray, if it exists in the termarray. Otherwise, the termarray is left intact.

For example, consider a termarray `ta` with elements 2X^2, 3X, 4 and Y^3.

```
term  t = 3X ;
ta = ta  - t ;
```

After this statement, `ta` will have 2X^2, 4, and Y^3 as its element. The element 3X has been removed from the termarray.

```
int i = 6;
ta = ta – i;
```

The execution of the above statements will result in `ta` being left unchanged, since `ta` does not have '6' as one of its elements. Please note that the statement `ta = ta – I` tries to remove the term 6 from the termarray. It does not subtract 6 from the existing '4' in `ta.` In short, when a termarray is involved in an '-' operation, the temarray has to be the minuend, and the subtrahend, if present, is removed or deleted from the

termarray thus reducing the length of the array.

When the operands are termarray and polynomial, the minuend has to be of type termarray. In such a case '-' will break poly into its constituent terms and remove these terms from termarray if it exists. Otherwise, the termarray is left intact.

For example, consider a termarray `ta` with the elements 2X^2, 3X, 4, Y^3 and 3Y^2.

```
Poly p = 2X^2 + 4;
ta = ta  - p;
```

After execution of the above statements, `ta` will have 3X, Y^3 and 3Y^2 as its elements. The elements 2X^2 and 4 were terms of the subtrahend poly, these terms were removed from `ta`.

Another example, consider termarray ta with elements 2X^2, 3X, 4, Y^3 and 3Y^2.

```
Poly p = 3Y^2 + 4Z ;
ta = ta – p;
```

After execution of the above statements, `ta` will have 2X^2, 3X and Y^3, 3Y^2 has been removed from `ta` since it was part of the subtrahend (`poly p`). 4Z which was part of `p` was not present in ta and so it does not affect the elements in `ta`.

Chararrays can be added and subtracted to chararrays and char datatypes. The behavior is the same as that of termarray addition and subtraction.

3.3.3.3 Relational - '<', '>', '<=', and '>='represent the less than, greater than, less than or equal to, and greater than and equal to relational operators, respectively. These operators are used to compare polynomials and terms and are all relational expressions whose return type is either a 0 or 1. Operators can be used in between expressions, polynomial, and terms.

Expression relational_op expression
poly relational_op term → (4X – 2Y – 2Z^3) < 3 (returns 0)
poly op poly → (2X) >= (4X -2X) (returns 1)

3.3.3.3.1 Equality – '==', '!=' are the equal to and not equal to operators, respectively. They have lower precedence than relational operators. Like

relational operators, a 0 or 1 is returned.

Expression equality_op expression
term equality_op poly → 3X == (4X+2X-3X) (returns 1)
poly equality_op poly → (4Y+2Y+1Y) != (8Y +0Y +10) (returns 1)

3.3.3.4 Power – The power operator, '^', is used to raise a variable to a particular degree. '^' must followed by an optional '+' or '-'and a mandatory float.

Variable power additive operator int → X^-3
float variable power operator int → 5X^11

3.4 DECLARATIONS

Declarations are used within the function definition to specify the interpretation of a particular identifier. Declarations have the form

declaration:
type-specifier declarator-list;

type-specifier:
poly
polyeq
int
float
term
termarray

The declarator-list appears in a declaration and is a sequence of comma separated declarators.

Declarator-list:
Declarator
Declarator , declarator-list

Declarator:
Identifier
Declarator ( )

(declarator)

Each declarator contains exactly one identifier, which is the identifier that is being declared. An identifier without a declarator has the type indicated by the type-specifer which heads the declaration where the identifier appears.

Examples of declaration:

int i , int k, j, poly p1, polyeq getequation(), termarray polyterms

3.5. STATEMENTS

Most statements are expression statements of the form:

Expression;

3.5.1 Compound Statement

Several statements can be used in place of one statement.

Compound-statement:
"begin" statement-list "end"

statement-list:
statement
statement statement-list

3.5.2 Conditional Statement

Two forms of conditional statement are:

If (expression) statement
If (expression) statement else statement

3.5.3 Loop statements

Two forms of loop statements are while and do while.

While (expression) statement end

do statement while (expression) end

3.5.4 Break statement

The break statement causes termination of the smallest enclosing while or do while statement. Control passes to the statement immediately after the end of the while or the do while statement.

Break;

3.5.5 Return statement

return;
return (expression);

A function returns to its caller by means of a return statement. In the first case no value is returned. This is the case when the function is declared as type void. In the second statement the value of the expression is returned to the caller of the function.

3.5.6 Print statement

print arg-list;

arg-list:
    expr
    expr arg-list

The print statement will accept a variable number of arguments until the semi-colon. It will then print each argument to the standard output on a single line. The print statement will automatically append a newline character to the standard output.

3.6 EXTERNAL DEFINITION

An external definition is given for a function. An external definition declares an identifier and it is type. Function definitions have the form as shown below.

Function – definition:
    Type-specifier function-declarator function body

Fuction-declarator:
    Declarator (parameter-list)
    Parameter-list:
        Identifier
        Identifier , parameter-list

Function-body
    Type-decl-list function-statement

Function-statement
    { declaration –list statement-list }

A simple example of a complete function definition:

```
func poly sumpoly(term t1, term t2)
begin
    vars
        poly p1;
    end
    p1 = t1 + t2;
```

```
        return p1;
end;
```

3.7 SCOPE RULES

There are two different kinds of scope – global scope and local scope.

3.7.1 – Global Scope

Global variables can be declared using the *vars* block **outside** a function definition. For example, this is sample PML code to declare variables in the global scope.

```
Vars
     poly p1;
     poly p2;
end

func void function1()
begin
     ... statements ...
end

vars
     poly p3;
     int i1;
end

func void function2()
begin
... statements ...
end
```

In this example, the variables p1, p2, p3 and i3 are declared in the global scope. Multiple *vars* blocks can be declared at the global scope. However, two global variables cannot share the same name/symbol, even in separate *vars* blocks. Functions can only be declared in the global scope. It is an error to declare a function inside another function.

All global variables are resident in memory from the moment the program runs until the program terminates. A global variable is considered in static scope from the line at which it was declared until the end of the file. In the previous code sample, function2() can make references to p1, p2, p3 and i1 – while function1() can only make references to p1 and p2.

3.7.2 – Local Scope

Variables can also be declared in PML using the *vars* block **inside** a function.

These variables are visible only inside the function, so it uses local scope. For example, the following two functions in PML contain local scope variables.

```
Func void function1()
begin
     vars
          poly p1;
          poly p2;
          int i1;
     end
     ... statements ...
end

func void function2()
begin
     ... statements ...

     vars
          poly p3;
          poly p1;  → OK
          float n1;
          int n1;  → Error
     end

     ... statements ...

     vars
          term p3;  → Error
     end

     ... statements ...
end
```

In this example, it is not an error to declare p1 in both function1() and function2(). This is because they are not within the same scope. It is an error to declare the integer n1 inside function2(), because n1 has already been declared in function2() as a float. It is also an error to declare the Term p3, even though the previous declaration of p3 is in a separate *vars* block.

Local scope variables can be declared at any part of the function. In the example above, the variables in function1() are declared at the top (before any statements). It is also possible, however, to declare a *vars* block in between statements, as seen in function2(). A function can also have multiple *vars* blocks, as seen in function2().

Every statement block introduces a new layer in the scope. A *vars* block can be used within a statement block. For example, consider the following PML code.

```
Func void function1()
```

```
begin
     #{ only global variables are valid }#

     vars
     poly p1;
     end
     ... statements ...

     #{ p1 and global variables are valid }#

     if (expr)
     begin
          ... statements ...

          #{ p1 and global variables are valid }#

          vars
               poly p2;
               int i3;
               term p1;  → Error
          end
          ... statements ...

          #{ p2, i3, p1 (Poly from previous }#
          #{ declaration) and global }#
          #{ variables are valid }#
     end

     #{ only p1 and global variables are}#
     #{ valid now}#

     ... statements ...

     vars
          term p2;  → OK
     end

     ... statements ...

     #{ p1, p2 and global variables are valid }#
end
```

In this example, a new statement block is created using the *if* construct. This introduces a new scoping layer, which sits on top of the parent scope. The same scoping semantics apply for statement blocks created using the *while* and *do ... while* constructs.

Local scope variables are resident in memory from the moment the *vars* block is declared until the “end” token for the corresponding statement block. The

comments in the code above describe these semantics for local scope.

Note that, unlike C/C++/Java, it is an error to declare Term p1 inside the *if* statement block, because p1 has already been declared as a Poly in the parent block. This is to prevent ambiguity when a reference is made to the p1 variable.

It is **not** an error to declare Term p2, even though p2 has been declared as a Poly inside the *if* statement block. This is because Poly p2 was no longer "visible" when Term p2 was declared.

Arguments to functions are also considered to be at the local scope. Consider the following example:

```
func void function1(poly p1, poly p2)
begin
     .. statements ...
end
```

The scoping rules for p1 and p2 are semantically similar to the scoping rules for p1 and p2 in this example:

```
func void function1()
begin
     vars
          poly p1;
          poly p2;
     end
     ... statements ...
end
```

If a function is called recursively, separate copies of the variables at the local scope will be pushed onto the stack and any references to these variables will use the copies on the top of the stack. When the function terminates, these variables will be popped off the stack and the previous variables will be used.

3.7.3 – Relationship Between Global and Local Scope

The general rule of thumb when declaring global or local variables is:

> *"If a symbol name is already statically visible at a certain scope, then it is an error to declare a variable using the same symbol name."*

This means it is an error to declare a variable at the local scope if the variable has already been declared at the global scope. It is **not** an error to declare a variable at the local scope even if it is declared later at the global scope. Consider the following code sample:

```
vars
```

```
     poly p1;  → OK
end

func void function1()
begin
     vars
          poly p2;  → OK
          term p1;  → Error
          poly p3;  → OK
     end
end

vars
     poly p2;  → OK
     int p1;  → Error
end

func void function2()
     vars
          poly p2;  → Error
          poly p3;  → OK
     end
end
```

Declaring p2 in function1() is not an error; however, declaring p2 in function2() is an error, because p2 has been declared at the global scope between function1() and function2().

3.8 NAMESPACE RULES

PML maintains two namespaces – the function namespace and the variable namespace. It is an error to declare two functions with the same name and the same list of arguments. However, it is not an error to declare two functions with the same name if they have a different list of arguments, implying that functions can be overloaded. It is also an error to declare two variables with the same name, if they are in the same scope (see section on "Scope Rules"). Variables and functions can share the same name. The parenthesis is used to resolve ambiguity between variables and functions.

3.9 ENTRY POINT

There is only one entry point to the program which is defined by a function called main(), that does not take an any arguments. The main function must exist in all programs. If main() is not found, an error message will be printed. The main() is guaranteed to be the first function executed in a PML program. The user is free to overload the main function; however, there should always be exactly one main function with no arguments. This main function with no arguments will be invoked by the interpreter after parsing and static

semantic checks are completed.

3.10 SEMANTICS FOR VARIABLE INITATION

Whenever a variable (local or global) is declared there is an optional initialization value. The semantics for performing this initialization is slightly different for local and global variables.

3.10.1 – Local Variables

This initialization procedure will be internally converted to an assignment statement that will be executed directly after the end of the *vars* block. Consider the following PML code:

```
func int init_i3()
begin
     return 1 + 1;
end

func void function1()
begin
     vars
          int i = 3;
          int i2 = i;
          int i3 = init_i3();
     end
end
```

Local variables can be initialized with the return value of a function (as seen with i3). This code will be converted internally to the following PML code:

```
func int init_i3()
begin
     return 1 + 1;
end

func void function1()
begin
     vars
          int i;
          int i2;
          int i3;
     end
     i = 3;
     i2 = i;
     i3 = init_i3();
end
```

3.10.2 Global Variables

The code conversion for local variables is relatively straight forward. However, the code conversion for global variables is a little more interesting. Consider the following code:

```
vars
     int i = 3;
     int i2 = i;
end

func void function1()
begin
     ... statements ...
end

vars
     int i3 = i2 + 5;
end
```

In this example, PML will create temporary "initializer functions" directly after the *vars* block. These initializer functions will be run during startup – before executing main(). So, the code above will be converted to something which will look like this:

```
vars
     int i;
     int i2;
end

func void @init1()
begin
     i = 3;
     i2 = i;
end

func void function1()
begin
     ... statements ...
end

vars
     int i3;
end

func void @init2()
begin
     i3 = i2 + 5;
```

```
end
```

Here, the '@' symbol is added as a prefix to the function name to ensure that there are no user-defined functions with the same name and also to ensure that the user will not call these functions. When running a PML program, the interpreter will first execute all functions beginning with '@' in the order in which they were added to the symbol table. After this, the interpreter will execute the main() function, as stated in the Section 10, "Entry Point".

The result is that the variables will be declared and initialized in the way that was expected by the programmer. Programmers should be aware that it is an error to initialize a global variable using a function. Consider the following PML code:

```
func int my_init()
begin
     return 1 + 1;
end

vars
     int i = my_init();  → Error
end

func void main()
begin
     ... statements ...
end
```

This example code will be converted to the following code by PML:

```
func int my_init()
begin
     return 1 + 1;
end

vars
     int i;
end

func void @init1()
begin
     i = my_init();  → Error
end

func void main()
begin
     ... statements ...
end
```

Based on the semantics described earlier, this code will execute my_init() before

main(), which is illegal. The PML interpreter guarantees that main() is always the first function that gets called (see Section 10 on “Entry Points”). Therefore, this PML code will just print an error message.

Chapter 4
Project Plan

4.1 Project Process

All team members in the group followed these processes during the various stages of development of the project.

4.1.1 Planning and Specification

General guidelines and specifications were planned during group meetings at different stages of the project life cycle. These meetings were extremely helpful in understanding the scope of the language, efficiency of implementation, setting up incremental goals, and division of labor. Specifications of interfaces between two or more components, for example, the parser and the AST, the AST and the background classes were laid out in these meetings so that parallel work can be achieved.

4.1.2 Development and Testing

Each team member was responsible for testing his or her work and also testing other people's work. This made each other aware of what the others were doing in the project and made it easier when we were putting it together. Each of the developed stages was subjected to unit testing. When ever any changes were made, the entire code base was subjected to integration and regression testing. Most of the major bugs were identified and reported during the unit and integration testing phases.

4.2 Team Responsibilities

Listed below are the primary responsibilities of each team member. A collaborative effort of all members was implored for testing, debugging, and documenting.

Team Member	Responsibility
Melinda Agyekum	Lexer and Parser
Shazan Baig	Tree Walker and Architecture
Hari Kurup	Compiler Back-end
Subadhra Sridharan	Lexer and Parser

4.3 Programming style (Coding Conventions)

4.3.1 ANTLR Conventions

ANTLR productions always followed the format of:

```
production:     <non-terminal> or <terminal>
                | non-terminal or <terminal>
               ;
```

If a multiple rule existed, the pipe symbols were always placed at the beginning of the statement. All semi-colons were placed on single lines.

The lexer token names are in upper case and the parser rules are in lower case.

4.3.2 JAVA Conventions

JAVA codes were written using standard programming conventions. The Hungarian notation and the Camel notation for all Java programs were used. The two notations were handy since members are often familiar with different standards and adapting temporarily to a different standard does not come easy.

- In Hungarian notation member variables start with ***m_*** followed by a character that indicates the type of the variable. *m_aTerms* indicates a member variable which is an array of terms.
- In the Camel notation, variables start with a three letter combination, in lower case, which indicates the type of the variable followed by a string. *intValue* indicates an integer variable.
- Member functions in all the Java code use the Camel notation. For example, *int ThisIsAFunction(int intA, int intB)*
- 'if' statement stands on a single line. The '{' '}' occupies an entire line
- The above convention also applies for all 'loop' and 'switch' statements
- 'goto' statements were not used
- The name of a Java class must relate to its function. For Example, the class PMLMath is the back-end function for math operations.

4.4 Project Timeline

The flowing deadlines were maintained for the completion of the project.

Date	Task
9-6-03	Language Topic and Features Confirmed
9-22-03	White Paper Submitted
10-3-03	Language Conventions and Environment Designed
10-18-03	Lexer and Parser Completed
10-28-03	Language Reference Manual

11-13-03	Tree Walker Completed
11-21-03	Back End Completed
12-9-03	Project Presentation
12-9-03	Project Demonstration
12-17-03	Final Paper Submitted

4.5 Software Development Environment

PML was developed using the Microsoft Windows operating system. The primary applications used to develop the project were JAVA SDE 1.4.1 and ANTLR 2.7.2. The source code was controlled using CVS, which permitted the integration of the Windows environment with files stored in a UNIX environment.

4.6 Project Log

A running log of all key dates were accounted for and are listed below.

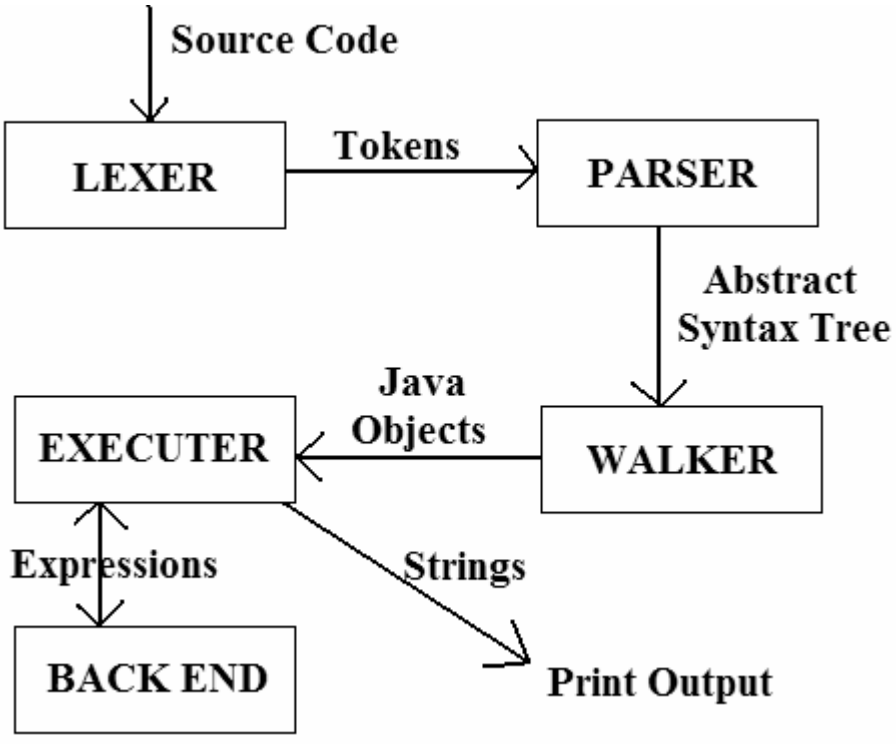
Date	Task
9-16-03	Language Topic and Features Confirmed
9-22-03	White Paper Submitted
10-3-03	Language Conventions and Environment Designed
10-18-03	Lexer and Parser Completed
10-28-03	Language Reference Manual
11-13-03	Tree Walker Completed
11-21-03	Backend Completed
11-22-03 - 12-6-03	Integrated & Regression Testing
12-6-03	Final Testing Phases
12-9-03	Project Presentation
12-9-03	Project Demonstration
12-17-03	Final Paper Submitted

Chapter 5
Architecture & Design

5.1 - Compiler Overview

The PML compiler was developed using ANTLR and JAVA. ANTLR was used primarily for the lexer and parser; the walker was written using a combination of ANTLR and embedded JAVA code. The executer (runtime environment) and the backend (mathematical functions) were written entirely in Java.

The diagram below shows a brief overview of the dataflow in the compiler.



5.2 - Lexer

The main job of the lexer is to convert the stream of characters into a source file into a stream of tokens. The lexer also removes all the whitespace, comments, and any other

characters that are not recognized by the parser. This makes it easier for the parser since the parser does not need to handle irrelevant characters.

Below is an example of some of the ANTLR code in the lexer for handling single-line and multi-line comments:

```
COMMENT : "#{"
         (
            options {greedy=false;}:
            (
               ('\r' '\n') => '\r' '\n' { newline(); }
               | '\r'                   { newline(); }
               | '\n'                   { newline(); }
               | ~('\n' | '\r')
             )
          )*
      "}#"
    { $setType(Token.SKIP); };

SL_COMMENT
        : "#"
        (~('\n'|'\r'|'{'))* ('\n'|'\r'('\n')?)
        {newline();}
        { $setType(Token.SKIP); }
        ;
```

5.3 - Parser

The parser takes the stream of tokens generated by the lexer and check to ensure that the tokens are in the correct order (i.e. check the syntax of the code). If the syntax is correct, then the parser generates an abstract syntax tree (AST), which would be passed to the walker.

The parser uses a greedy algorithm to match a rule with the longest stream of tokens possible. An example of ANTLR code for the “if” statement rule is:

```
ifstmt : "if"^ LPAREN! cond_expr RPAREN! statement
         (options {greedy=true;} : "else"! statement)?
         ;
```

5.4 - Walker & Executer

The walker processes the AST which is passed from the parser, and it will then generate a bunch of Java objects that will be used by the executer. These Java objects can be thought of as more specific syntax trees. This means that there are special classes that handle different kinds of statements, for example PMLStatementIf, PMLStatementAssign and a

few others.

The design of these classes was done using an object-oriented approach, which utilized interfaces, classes and polymorphism. This makes it very easy to expand the language to support many other kinds of statements or expressions.

The mapping of the AST to the Java objects is quite simple. An example of sample ANTLR code with embedded Java that will generate a PMLStatement from the 'if' statement defined in section 5.3 is:

```
ifstmt returns [PMLStatementIf r] throws Exception
{
    r = null;
    PMLCondition cond;
    PMLStatement thenPart;
    PMLStatement elsePart = null;
}
    : #("if"
        cond = pred:cond_expr
        {
            AST astThen = pred.getNextSibling();
            AST astElse = astThen.getNextSibling();
            thenPart = statement(astThen);
            if (astElse != null)
                elsePart = statement(astElse);
            r = new PMLStatementIf(cond, thenPart,
                                   elsePart);
        }
        );
```

The following sections will briefly explain some of the Java interfaces and classes that are used to execute PML code.

5.4.1 - class PMLRuntimeEnvironment

This is the main class used by the executer. Throughout the execution of a PML program, there is only one instance of this class. It keeps track of the function stack and also contains an array of compilation units (files). This class allows form multiple files to be linked together and also locates external functions to execute them.

5.4.2 - class PMLCompilationUnit

Every time the PML compiler compiles a source file, it a PMLCompilationUnit object is generated and added to the runtime environment. The PMLCompilationUnit contains all the global variables (PMLVariable) and all the functions (PMLFunction) in a file.

5.4.3 - abstract class PMLVariable

abstractclassPMLVariable is the base class for all variables in PML. It contains several functions that are quite useful:

setValue()- This function will set the value of this PMLVariable to another PMLVariable. This function modifies the variable and also implements all the conversion and typecasting rules as defined in the LRM.

compare()- This function will compare the value in this PMLVariable with another PMLVariable and return true if they are identical. It is used primarily by the backend when trying to determine like/unlike terms.

negate()- This function will return a negative version of itself. It is used by the backend when performing subtractions.

The sub-classes of PMLVariable are: VoidVariable, CharVariable, CharArray, IntVariable, FloatVariable, TermVariable, TermArray and PolyVariable.

5.4.4 - class PMLFunction

This class represents a function in PML. It consists of a return type, arguments, and an array of statements. It also contains a flag to specify whether it is a built-in function, since built-in functions and user-defined functions are represented in the same way by the executer.

5.4.5 - interface PMLSymbolTable

This interface is implemented by classes that track and locate variable declarations while walking and executing the tree. It exposes the following functions:

addVariable() - This function will add a variable to the symbol table.

findVariable() - This function will check the symbol table for a variable and return a reference to the variable (if found). If not, it will check the parent symbol table for the variable.

setParent() - This function is used to set the parent of the symbol table. This allows us to implement multiple levels of scope in our language.

The classes that implement this interface are: PMLCompilationUnit, PMLFunction and PMLStatementBlock.

5.4.6 - class PMLFunctionInstance

This class is analogous to an activation-frame in compiler terminology. It is used to track the return value (a PMLVariable) and also maintains the stack for local variables.

5.4.7 - interface PMLStatement

This interface is implemented by all the statement classes. It exposes a single function:

runStatement() - This function will execute the statement.

The classes that implement this interface are:
PMLStatementAssign, PMLStatementBlock, PMLStatementBreak, PMLStatementDo,

PMLStatementExpr, PMLStatementPrint, PMLStatementReturn, PMLStatementWhile, PMLStatementIf.

5.4.8 - interface PMLExpression

This interface is implemented by all classes that represent expressions in the language. It exposes a single function:

> ***evaluate()*** - This function will evaluate the expression and return a PMLVariable object.

The classes that implement this interface are:
PMLExprArith, PMLExprFuncCall, PMLExprLiteral, PMLExprTypeCast, PMLExprVariable, and PMLExprVariableArray.

5.5 - Backend

The backend is responsible for all the mathematical functions supported in PML. This involved the addition, subtraction, multiplication and division of expressions. The entire backend is embodied in a Java class called PMLMath. This class contains a set of functions that are used to perform these arithmetic operations with PML variables. The behavior of these functions follow the rules of the PML language, as described in the Language Reference Manual *(see section 3.4.3 'Operators')*.

The primary function in PMLMath.java is called “PlusOp_General”. This function is called from the executer (from PMLExprArith.java) and it can handle any kind of PML arithmetic operation defined in the LRM. The “PlusOp_General” function branches out to other more specialized functions, depending on the types of the arguments passed to it.

Chapter 6
Testing Plan

The testing for PML was split the testing into three stages – unit testing, integrated testing and regression testing. Each stage helps in weeding out flaws that the previous stage may not have caught.

6.1 Unit Testing

Unit testing is generally seen as a "white box" test class which is biased to looking at and evaluating the code as implemented, rather than evaluating conformance to some set of requirements. For PML, unit testing was performed on the major modules of the lexer, parser, walker, executor, and the back-end. Each of these modules were tested separately with sufficient test cases to satisfy their behavior matched the code as implemented.

6.2 Integrated Testing

Once each module has been satisfactorily tested, the next step was to combine these modules into a single working unit and test it. The lexer and parser were in a single unit of code, integrated with the walker and executor. At this time, limited features of PML were implemented and the entire unit was tested. The back-end was integrated as the next step.

Once integration was complete, the test cases were written. Because PML deals with symbolic mathematics, the test cases often consisted of symbolic operations. But there were other test cases which were used to test certain features of the language, like recursive function calls.

6.3 Regression Testing

In most software systems integrated testing reveals problems with the system. PML was no exception and several problems were detected from integrated testing. Once the fixes were made, all test cases were rerun through the system to ensure that no new bugs were introduced and the known problems have been fixed.

As is the nature of software development, testing and fixing is a cycle. It is an on-going process, the ultimate aim of which is improve the quality of the software.

Chapter 7
Lessons Learned

As with any project, there is always scope for learning something useful for the future. In our case we have learned a few lessons that will be useful for the future.

7.1 Quotes

Each member of the PML team would like to share a few thoughts and words of wisdom about our experience on the project.

Melinda Agyekum – "*Organization, team work, and a sound understanding of what is being implemented are the most important aspects for implementing a successful project. Although work is divided among all members it is important for all to maintain a general understanding of what other teammates are working on. This allows for all members of the team to contribute any ideas or assistance which is needed. I contribute the success of our project to the open lines of communication and the general respect that each team member held for one another. Overall this was a very good, exciting, and practical project and to be a part of*".

Shezan Baig - "*One of the key issues involved while developing this compiler was coordination among team members. We didn't start using CVS right at the beginning of the project, but once we started, it made life a lot easier. Also, we think it's pretty cool to be able to split the compiler into separate chunks that could be developed in parallel. This helped speed up the process quite a bit. If I were to continue working on this project, I would like to improve the error reporting module (currently the compiler just prints the error and exits).Possibly I would like to give programs a method to recover from errors, for example implementing a try...catch block structure. I don't imagine this will be particularly hard to do, given the modularity of our current design."*

Hari Kurup – "*Enough time should be spent on a proper and detailed study of the domain for the project. All possible cases that may fall into the scope of the project must be considered. This affects the way certain data is represented in the system, which in turns decides the data structure used for it. In our case we should have spent time discussing the different forms of polynomials that needs to be represented, and should come up with a representation that was more flexible. The current methodology of representation is somewhat restrictive though its serves its purpose for this project.*"

"*The division of labor was very helpful in speeding up the progress of the project. The*

fact that documentation and testing was distributed to all team members also enabled in getting more team input."

Subadhra Sridharan – *"I learnt that team work plays an important role in the successful completion of the project work. We as a team worked well and hence we were able to meet our goals early. We planned meetings once or twice a week and hence everybody was aware of what was going on. We learnt about CVS when we were getting overwhelmed with the emails we were sending each other, and it was getting difficult to keep track of which was the latest version of files. But we caught ourselves in the initial stages and set up CVS and everything went extremely well after this. The division of labor was done taking into consideration the convenient meeting times for people who needed to directly interact and as a result we were able to achieve parallel work to the fullest extent. I also learnt that if you are given a mentor who is willing to spend time with you on the project, then exploit this to the fullest extent. We set up frequent meetings with our mentor and as a result were able to get concepts clarified or doubts cleared as early as possible."*

7.2 Future Plans

Additional areas which we would like to enhance in the future include:

- Improve error reporting module by reporting errors messages that are more descriptive.
- A method to recover from errors, for example implementing a try...catch block structure.
- In-built functions that will enable writing simpler polynomial multiplication and division programs in PML.
- Integration tables that will enable writing symbolic integration algorithms.

APPENDIX A
Standard Library Functions

This section is work in progress. More functions will be added if necessary during the course of the development of this language.

The following functions deal with polynomials and terms.

Term coeff (...) : This function takes a term as a parameter and returns the coefficient of that term. Parameters of type int and float are considered as terms with no variables, and so the coefficient of such a term is the term itself. Acceptable invocations of this function are as follows:

coeff(term t)
coeff(int i)
coeff(float f)

examples: `term t = 2X^2;  coeff(  t );` ***ReturnValue*** 2
`term t = aX^2 ; coeff( t );` ***Return Value*** a

term degree(…) : This is an overloaded function and so it can accepts different number and types of parameters. Essentially, this function returns the total degree of a term or polynomial. The overloading feature can be used to specify the variable whose degree is expected. Valid invocations of the function are:

degree(term)
degree(poly)
degree(term , char)

examples: `term t = 3X^3; degree( t );` ***Return Value*** 3
`term t = 2Y^3Y^2; degree( t );` ***Return Value*** 5
`term t = 2X^3Y^2; degree ( t , 'Y' );` ***Return Value*** 2
`poly p = 3X^2 + 4X^5Y^3 + Y ;  degree( p );` ***Return Value*** 8
`term t = 3X^3; degree (t, 'Y');` ***Return Value*** 0

termarray polyterm(...) : This function takes a parameter and returns a termarray with the parameter as a member of the array. The input parameter will be broken into constituent terms, if it happens to be a polynomial. Parameters of type int and float are considered as terms with no variables. Valid invocations of this function are as follows:

polyterm (term t);
polyterm (int i);
polyterm (float f);
polyterm (char c);

polyterm (poly p);

examples: `int i = 20; polyterm( i ) ;`
Return Value : a termarray of length 1, with 20 as its element

`term t = 4X^3; polyterm ( t );`
Return Value: a termarray of length 1, with 4x^3 as its element.
`poly p = 2X^2 + 3X + 4; polyterm( p );`
Return Value: a termarray of length 3, with 2X^2, 3X and 4 as its elements.

int length (...) : This function takes a termarray and returns an integer that represents the number of items in the array. Valid invocations of the function:

length(terarray ta)
examples: `termarray ta;`
```
poly p = 2x^3 + 3x^2 + 4x;
ta = polyterm( p );
length ( ta );
```
Return Value 3

chararray variable(...): This function takes a term and returns all the variables in that term as elements of a chararray. Valid invocations of this function:

term t1 = 2XY;
chararray ca;
ca = variable(t1);
Return Value: a chararray with X and Y as its 2 elements.

APPENDIX B
Source Code

B.1 ANTLR CODE

PML.g (Parser & Lexer)

```
class PMLParser extends Parser;
options { buildAST = true; k = 3; }

start  : (function|vars_block)* EOF!
       ;


vars_block : "vars"^ (var_decl)* "end"!
           ;

var_decl : datatype ID^ (EQSIGN! expr)? SEMI!
         ;


expr : polyexpr
     | typecast_expr
     ;


typecast_expr : LPAREN! datatype RPAREN! expr
    { #typecast_expr = #([TYPECAST_EXPR, "TYPECAST_EXPR"], #typecast_expr); }
    ;

// make sure in walker that ID only contains 1 char
char_literal : QUOTE! (ID|LETTER) QUOTE!
             { #char_literal = #([CHAR_LITERAL, "CHAR_LITERAL"], #char_literal); }
    ;

funccall_stmt : funccall SEMI!
    ;

arglist : (arg (COMMA! arg)* )?
    ;

arg : datatype ID^
    ;


funccall : ID LPAREN! (expr (COMMA! expr)*)? RPAREN!
            { #funccall = #([FUNCCALL, "FUNCCALL"], funccall); }
    ;

function   : "func"^ datatype ID LPAREN! arglist RPAREN! statement_block ((SEMI)?)!
           ;

statement : assignstmt
           | ifstmt
           | whilestmt
           | dostmt
           | returnstmt
           | breakstmt
           | vars_block
           | funccall_stmt
           | print_stmt
           | statement_block
           | SEMI!
           ;

print_stmt  : "print"^ (string_const|expr) (COMMA! (string_const|expr))* SEMI!
    ;
```

```
string_const : STRING_CONSTANT^;

statement_block : "begin"^ (statement)* "end"
           ;


assignstmt : lvalue EQSIGN! expr SEMI!
            { #assignstmt = #([ASSIGNMENT, "ASSIGNMENT"], #assignstmt); }
           ;

lvalue : ID^ (tarray)?
    ;


cond_expr : expr (GREATER^ | LESSTHAN^ | GREATEQ^ | LESSEQ^ | CONDEQ^ | NOTEQ^) expr
              ;

ifstmt    : "if"^ LPAREN! cond_expr RPAREN! statement
          (options {greedy=true;} : "else"! statement)?
          ;

whilestmt : "while"^ LPAREN! cond_expr RPAREN! statement
          ;

dostmt    : "do"^ statement "while"! LPAREN! cond_expr RPAREN! SEMI!
          ;

returnstmt : "return"! (expr)? SEMI!
            { #returnstmt = #([RETURN_STMT, "RETURN_STMT"], #returnstmt); }
           ;

breakstmt : "break"^ SEMI
          ;


datatype : "int"
     | "float"
     | "poly"
     | "term"                // added by Shezan
     | "termarray"        // added by Shezan
     | "char"
     | "poly_equ"
     | "void"
     | "chararray"      // added by Hari
         ;


poly_equ: polyexpr (EQSIGN|GREATER|GREATEQ|LESSTHAN|LESSEQ) polyexpr
        ;


/* Added by Subadhra */

polyexpr
    : poly_multdiv ((PLUS|MINUS) poly_multdiv)*
          { #polyexpr = #([POLYEXPR, "POLYEXPR"], #polyexpr); }
    ;

poly_multdiv
    : term ((DIV|MULT) term)*
          { #poly_multdiv = #([POLYEXPR, "POLYEXPR"], #poly_multdiv); }
    ;


term : ffloat (letter_degree)*
        { #term = #([TERM_FLOAT, "TERM_FLOAT"], #term); }
    | (letter_degree)+
        { #term = #([TERM_LETTER, "TERM_LETTER"], #term); }
    | id_ref (letter_degree)*
        { #term = #([TERM_ID, "TERM_ID"], #term); }
    | LPAREN! expr RPAREN! (letter_degree)*
        { #term = #([TERM_PAREN, "TERM_PAREN"], #term); }
    | funccall (letter_degree)*
        { #term = #([TERM_FUNCCALL, "TERM_FUNCCALL"], #term); }
```

```
    | char_literal (letter_degree)*
        { #term = #([TERM_CHAR, "TERM_CHAR"], #term); }
        ;

letter_degree
    : LETTER^ (degree)?
    ;

degree
    : CARROT! ffloat
        { #degree = #([DEGREE_FLOAT, "DEGREE_FLOAT"], #degree); }
    | CARROT! char_literal
        { #degree = #([DEGREE_CHAR, "DEGREE_CHAR"], #degree); }
    | CARROT! id_ref
        { #degree = #([DEGREE_ID, "DEGREE_ID"], #degree); }
    | CARROT! LPAREN! expr RPAREN!
        { #degree = #([DEGREE_PAREN, "DEGREE_PAREN"], #degree); }
    | CARROT! funccall
        { #degree = #([DEGREE_FUNCCALL, "DEGREE_FUNCCALL"], #degree); }
    ;

id_ref
    : ID^ (tarray)?
    ;

/* Changed by Subadhra. Float can look like 0.342 or -2.320, +0.543 or -4.0*/
ffloat  : (PLUS!)? INT^ (DOT INT)?
        | MINUS^ INT (DOT INT)?
        ;

tarray  : LBRACK! expr RBRACK!           //Examples:        [13]
        ;


/*LEXICAL ANALYZER*/
class PML_LEXER extends Lexer;

options { testLiterals = false; k = 2; charVocabulary ='\3'..'\377';}

QUOTE : '\'';
DOT     : '.';
CARROT        : '^' ;
COMMA        : ',' ;
CONDEQ        : "==";
DIV        : '/' ;
EQASSIGN: "=>";
EQSIGN  : '=' ;
LBRACE        : '{';
LBRACK        : '[';
LESSEQ        : "<=";
LESSTHAN: '<';
LPAREN  : '(' ;
MINUS        : '-' ;
MULT        : '*' ;
GREATEQ        : ">=";
GREATER        : '>';
NOTEQ        : "!=";
PLUS        : '+' ;
RBRACE        : '}';
RBRACK        : ']';
RPAREN  : ')' ;
SEMI        : ';';


STRING_CONSTANT : '"'! ( ~('"' | '\n') | ('"'! '"'))* '"'! ;

LETTER: ('A'..'Z');

protected SMALL_LET: ('a'..'z');

protected DIGIT : '0'..'9';

INT : (DIGIT)+;

ID options { testLiterals = true; }
```

```
    : SMALL_LET (SMALL_LET | DIGIT | '_')* ;

WS        : (' ' | '\t' | '\n' { newline(); } | '\r' )
          { $setType(Token.SKIP); } ;


/* Changed by Subadhra single line comment starts with # */
COMMENT        : "#{"
           //('{') => '{'
                 (//Prevent .* from eating the whole file
                  options {greedy=false;}:
                  (
                    ('\r' '\n') => '\r' '\n' { newline(); }
                    | '\r'                   { newline(); }
                    | '\n'                   { newline(); }
                    | ~('\n' | '\r')
                  )
                 )*
                 "}#"

        { $setType(Token.SKIP); }
        ;

SL_COMMENT
        : "#"
        (~('\n'|'\r'|'{'))* ('\n'|'\r'('\n')?)
        {newline();}
        { $setType(Token.SKIP); }
        ;
```

PMLWalker.g

```
{
import java.util.*;
}
/* 27-Nov-2003 Hari Kurup : added chararray */
/* 29-Nov-2003 Shezan : added typecasting */

class PMLWalker extends TreeParser;

options { k = 2; }

{
    PMLCompilationUnit pcu;
    Stack pSymbolTableStack;
    PMLFunction pCurrentFunction;
    int nLoopCounter;
}

begin returns [PMLCompilationUnit r] throws Exception
{
    nLoopCounter = 0;
    int nInitFuncCounter = 0;
    r = pcu = new PMLCompilationUnit();
    pSymbolTableStack = new Stack();
    pSymbolTableStack.push(pcu);

    PMLStatementBlock vars_blockRet;
}
    :
    (

     function
     | vars_blockRet = vars_block
       {
            PMLFunction initFunc = new PMLFunction();
            initFunc.m_sFunctionName = "@init"
                                     + Integer.toString(nInitFuncCounter);
            initFunc.m_statementBlock = vars_blockRet;
            initFunc.setParent(pcu);
            pcu.addFunction(initFunc);
            nInitFuncCounter++;
```

```
       }

    )*

    {
        pSymbolTableStack.pop();
    }
    ;

vars_block returns [PMLStatementBlock blockInit] throws Exception
{
    blockInit = new PMLStatementBlock();
    Vector var_declRet;
}
    : #("vars"
        (
            var_declRet=var_decl
            {

((PMLSymbolTable)pSymbolTableStack.peek()).addVariable((PMLVariable)var_declRet.get(0));
                if (var_declRet.size() == 2)
                {
                    blockInit.addStatement((PMLStatement)var_declRet.get(1));
                }
            }
        )*
      )

    ;

var_decl returns [Vector r] throws Exception
{
    String var_type;
    r = new Vector();
    PMLVariable var;
    PMLStatementAssign ass = null;
}
    : #(ID var_type=pred:datatype
         {
            AST varName = #ID;
            var = PMLVariable.createVariable(var_type, varName.getText());
            r.add(var);

            AST assign = pred.getNextSibling();
            if (assign != null)
            {
                ass = new PMLStatementAssign(new PMLExprVariable(varName.getText()),
expr(assign));
                r.add(ass);
            }
         }
      )

;

expr returns [PMLExpression r] throws Exception
{
    r = null;
}
    : r = polyexpr
    | r = typecast_expr
    ;

typecast_expr returns [PMLExpression r] throws Exception
{
    r = null;
    String t;
}
    : #(TYPECAST_EXPR
        {
            AST astDataType = #TYPECAST_EXPR.getFirstChild();
            t = datatype(astDataType);
            AST astExpr = astDataType.getNextSibling();
            r = new PMLExprTypecast(PMLVariable.createVariable(t, "tmp"), expr(astExpr));
        }
```

```
        )
    ;

char_literal returns [PMLVariable r] throws Exception
{
    r = null;
}
    : #(CHAR_LITERAL
        {
            AST astChar = #CHAR_LITERAL.getFirstChild();
            if (astChar.getText().length() != 1)
                throw new Exception("Only single characters allowed between quotes!");
            CharVariable var = new CharVariable("tmp");
            var.m_cValue = astChar.getText().charAt(0);
            r = var;//new PMLExprLiteral(var);
        }
        )
    ;

polyexpr returns [PMLExpression r] throws Exception
{
    r = null;
}
    : #(POLYEXPR
          {
            AST astLeft = #POLYEXPR.getFirstChild();
            PMLExpression left = polyexpr(astLeft);

            AST astOp = astLeft.getNextSibling();
            if (astOp != null)
            {
                while (astOp != null)
                {
                    char op = astOp.getText().charAt(0);
                    AST astRight = astOp.getNextSibling();
                    PMLExpression right = polyexpr(astRight);
                    r = new PMLExprArith(left, op, right);
                    left = r;
                    astOp = astRight.getNextSibling();
                }
            } else
                r = left;
          }
        )
    | r = term
    ;

mul_div returns [PMLExpression r] throws Exception
{
    r = null;
}
    : #(MUL_DIV
          {
            AST astLeft = #MUL_DIV.getFirstChild();
            PMLExpression left = term(astLeft);

            AST astOp = astLeft.getNextSibling();
            if (astOp != null)
            {
                char op = astOp.getText().charAt(0);
                AST astRight = astOp.getNextSibling();
                PMLExpression right = mul_div(astRight);
                r = new PMLExprArith(left, op, right);
            } else
                r = left;
          }
        )
    ;

term returns [PMLExpression r] throws Exception
{
    r = null;
    PMLVariable var;
}
    : #(TERM_CHAR
```

```
    {
        AST astChar = #TERM_CHAR.getFirstChild();
        AST astLetter = astChar.getNextSibling();
        if (astLetter != null)
        {
            TermVariable varTerm = new TermVariable("termLiteral");
            var = char_literal(astChar);
            varTerm.m_fCoeff = new PMLExprLiteral(var);
            while (astLetter != null)
            {
                varTerm.m_aLetterDegrees.add(letter_degree(astLetter));
                astLetter = astLetter.getNextSibling();
            }

            r = new PMLExprLiteral(varTerm);
        } else
        {
            var = char_literal(astChar);
            r = new PMLExprLiteral(var);
        }
    }
    )
| #(TERM_FLOAT
    {
        AST astFloat = #TERM_FLOAT.getFirstChild();
        AST astLetter = astFloat.getNextSibling();
        if (astLetter != null)
        {
            TermVariable varTerm = new TermVariable("termLiteral");
            var = ffloat(astFloat);
            varTerm.m_fCoeff = new PMLExprLiteral(var);
            while (astLetter != null)
            {
                varTerm.m_aLetterDegrees.add(letter_degree(astLetter));
                astLetter = astLetter.getNextSibling();
            }

            r = new PMLExprLiteral(varTerm);
        } else
        {
            var = ffloat(astFloat);
            r = new PMLExprLiteral(var);
        }
    }
    )
| #(TERM_LETTER
    {
        AST astLetter = #TERM_LETTER.getFirstChild();
        TermVariable varTerm = new TermVariable("termLiteral");
        var = new IntVariable("coeffterm");
        var.setValue("1");
        varTerm.m_fCoeff = new PMLExprLiteral(var);
        while (astLetter != null)
        {
            varTerm.m_aLetterDegrees.add(letter_degree(astLetter));
            astLetter = astLetter.getNextSibling();
        }

        r = new PMLExprLiteral(varTerm);
    }
    )
| #(TERM_ID
    {
        AST astIDRef = #TERM_ID.getFirstChild();
        PMLExpression idref = id_ref(astIDRef);
        AST astLetter = astIDRef.getNextSibling();
        if (astLetter == null)
        {
            r = idref;
        } else
        {
            TermVariable varTerm = new TermVariable("termLiteral");
            varTerm.m_fCoeff = idref;
            while (astLetter != null)
            {
```

```
                    varTerm.m_aLetterDegrees.add(letter_degree(astLetter));
                    astLetter = astLetter.getNextSibling();
                }

                r = new PMLExprLiteral(varTerm);
            }
        }
        )
    | #(TERM_PAREN
        {
            AST astExpr = #TERM_PAREN.getFirstChild();
            PMLExpression pmlExpr = expr(astExpr);
            AST astLetter = astExpr.getNextSibling();
            if (astLetter == null)
            {
                r = pmlExpr;
            } else
            {
                TermVariable varTerm = new TermVariable("termLiteral");
                varTerm.m_fCoeff = pmlExpr;
                while (astLetter != null)
                {
                    varTerm.m_aLetterDegrees.add(letter_degree(astLetter));
                    astLetter = astLetter.getNextSibling();
                }

                r = new PMLExprLiteral(varTerm);
            }
        }
        )
    | #(TERM_FUNCCALL
        {
            AST astFuncCall = #TERM_FUNCCALL.getFirstChild();
            PMLExpression pmlExpr = funccall(astFuncCall);
            AST astLetter = astFuncCall.getNextSibling();
            if (astLetter == null)
            {
                r = pmlExpr;
            } else
            {
                TermVariable varTerm = new TermVariable("termLiteral");
                varTerm.m_fCoeff = pmlExpr;
                while (astLetter != null)
                {
                    varTerm.m_aLetterDegrees.add(letter_degree(astLetter));
                    astLetter = astLetter.getNextSibling();
                }

                r = new PMLExprLiteral(varTerm);
            }
        }
        )
    ;

letter_degree returns [LetterDegree r] throws Exception
{
    r = null;
}
    : #(LETTER
        {
            PMLExpression power;
            if (#LETTER.getFirstChild() != null)
            {
                power = degree(#LETTER.getFirstChild());
            } else
            {
                PMLVariable var = new IntVariable("tmp");
                var.setValue("1");
                power = new PMLExprLiteral(var);
            }
            r = new LetterDegree(#LETTER.getText().charAt(0), power);
        }
        )
    ;
```

```
degree returns [PMLExpression r] throws Exception
{
    r = null;
}
    : #(DEGREE_FLOAT
        {
            PMLVariable var = ffloat(#DEGREE_FLOAT.getFirstChild());
            r = new PMLExprLiteral(var);
        }
        )
    | #(DEGREE_CHAR
        {
            PMLVariable var = char_literal(#DEGREE_CHAR.getFirstChild());
            r = new PMLExprLiteral(var);
        }
        )
    | #(DEGREE_ID
        {
            r = id_ref(#DEGREE_ID.getFirstChild());
        }
        )
    | #(DEGREE_PAREN
        {
            r = expr(#DEGREE_PAREN.getFirstChild());
        }
        )
    | #(DEGREE_FUNCCALL
        {
            r = funccall(#DEGREE_FUNCCALL.getFirstChild());
        }
        )
    ;

id_ref returns [PMLExpression r] throws Exception
{
    PMLVariable var;
    PMLExpression index;
    r = null;
}
    : #(ID
        {
            var = ((PMLSymbolTable)pSymbolTableStack.peek()).findVariable(#ID.getText());
            if (var == null)
                throw new Exception("Undeclared identifier: '" + #ID.getText() + "'!");

            if (#ID.getFirstChild() != null)
                index = expr(#ID.getFirstChild());
            else
                index = null;

            if (index == null)
            {
                r = new PMLExprVariable(#ID.getText());
            } else
            {
                if (var.getVariableType() != PMLVariable.typeTermArray
                    && var.getVariableType() != PMLVariable.typeCharArray)
                {
                    throw new Exception("'" + #ID.getText() + "' is not an array. Cannot
use '[]' operator!");
                }

                r = new PMLExprVariableArray(#ID.getText(), index);
            }
        }
        )
    ;

funccall returns [PMLExprFuncCall r] throws Exception
{
    r = null;
    String sFunctionName;
    Vector t_aParams = new Vector();
    PMLExpression pExpr;
}
```

```
    : #(FUNCCALL
        {
            AST funcName = #FUNCCALL.getFirstChild();
            sFunctionName = funcName.getText();

            AST paramExpr = funcName.getNextSibling();
            while (paramExpr != null)
            {
                pExpr = expr(paramExpr);
                t_aParams.add(expr(paramExpr));
                paramExpr = paramExpr.getNextSibling();
            }

            r = new PMLExprFuncCall(sFunctionName, t_aParams);
        }
      )
    ;

ffloat returns [PMLVariable r] throws Exception
    { r = null; }
    : #(INT
        {
         String s = #INT.getText();
         AST dot = #INT.getFirstChild();
         if (dot != null)
         {
             s = s + dot.getText();
             AST fraction = dot.getNextSibling();
             if (fraction != null)
                s = s + fraction.getText();
            r = PMLVariable.createVariable(PMLVariable.typeFloat, "literalValue");
         } else
         {
            r = PMLVariable.createVariable(PMLVariable.typeInt, "literalValue");
         }
         r.setValue(s);
        }
      )
    | #(MINUS
        {
            String s = #MINUS.getText();
            AST intpart = #MINUS.getFirstChild();
            s = s + intpart.getText();
            AST dot = intpart.getNextSibling();
            if (dot != null)
            {
                s = s + dot.getText();
                AST fraction = dot.getNextSibling();
                if (fraction != null)
                    s = s + fraction.getText();
                r = PMLVariable.createVariable(PMLVariable.typeFloat, "literalValue");
            } else
            {
                r = PMLVariable.createVariable(PMLVariable.typeInt, "literalValue");
            }
            r.setValue(s);
        }
      )
    ;

statement_block returns [PMLStatementBlock r] throws Exception
{
  r = new PMLStatementBlock();
  r.setParent((PMLSymbolTable)pSymbolTableStack.peek());
  PMLStatement pStmt;
}
    : #("begin"

        {
            pSymbolTableStack.push(r);
        }

        (
        pStmt = statement
            {
```

```
            r.addStatement(pStmt);
            }
        )*

        "end"
        {
            pSymbolTableStack.pop();
        }

        )
    ;

cond_expr returns [PMLCondition r] throws Exception
{
    r = null;
    PMLExpression pExprLeft, pExprRight;
    int op=0;
}
    : (#(GREATER { op = PMLCondition.opGreater; } pExprLeft=expr pExprRight=expr)
     | #(LESSTHAN { op = PMLCondition.opLess; } pExprLeft=expr pExprRight=expr)
     | #(GREATEQ { op = PMLCondition.opGreaterEqual; } pExprLeft=expr pExprRight=expr)
     | #(LESSEQ { op = PMLCondition.opLessEqual; } pExprLeft=expr pExprRight=expr)
     | #(CONDEQ { op = PMLCondition.opEqualTo; } pExprLeft=expr pExprRight=expr)
     | #(NOTEQ { op = PMLCondition.opNotEqualTo; } pExprLeft=expr pExprRight=expr))

     {
        r = new PMLCondition(pExprLeft, op, pExprRight);
     }
    ;

statement returns [PMLStatement r] throws Exception
{
    r = null;
}

    : r = vars_block
    | r = statement_block
    | r = print_stmt
    | r = returnstmt
    | r = ifstmt
    | r = whilestmt
    | r = dostmt
    | r = funccall_stmt
    | r = assignstmt
    | r = breakstmt
    ;

breakstmt returns [PMLStatementBreak r] throws Exception
{ r = new PMLStatementBreak(); }
    : #("break"
        {
            if (nLoopCounter == 0)
                throw new Exception("'break' can only be used inside a loop!");
        }
        )
    ;

assignstmt returns [PMLStatementAssign r] throws Exception
{
    r = null;
}
    : #(ASSIGNMENT
        {
            AST astID = #ASSIGNMENT.getFirstChild();
            r = new PMLStatementAssign(lvalue(astID), expr(astID.getNextSibling()));
        }
        )
    ;

lvalue returns [PML_LValue r] throws Exception
{
    PMLVariable var;
    PMLExpression index;
    r = null;
}
```

```
    : #(ID
        {
            var = ((PMLSymbolTable)pSymbolTableStack.peek()).findVariable(#ID.getText());
            if (var == null)
                throw new Exception("Undeclared identifier: '" + #ID.getText() + "'!");

            if (#ID.getFirstChild() != null)
                index = expr(#ID.getFirstChild());
            else
                index = null;

            if (index == null)
            {
                r = new PMLExprVariable(#ID.getText());
            } else
            {
                if (var.getVariableType() != PMLVariable.typeTermArray
                    && var.getVariableType() != PMLVariable.typeCharArray)
                {
                    throw new Exception("'" + #ID.getText() + "' is not an array. Cannot
use '[]' operator!");
                }

                r = new PMLExprVariableArray(#ID.getText(), index);
            }
        }
        )
    ;


dostmt returns [PMLStatementDo r] throws Exception
{
    r = null;
    PMLCondition cond;
    PMLStatement stmt;
}
    : #("do"
        {
            nLoopCounter++;
        }
        stmt = pred:statement
        {
            nLoopCounter--;
            AST astCond = pred.getNextSibling();
            cond = cond_expr(astCond);
            r = new PMLStatementDo(cond, stmt);
        }
        )
    ;

whilestmt returns [PMLStatementWhile r] throws Exception
{
    r = null;
    PMLCondition cond;
    PMLStatement stmt;
}
    : #("while"
        cond = pred:cond_expr
        {
            nLoopCounter++;
            AST astStmt = pred.getNextSibling();
            stmt = statement(astStmt);
            r = new PMLStatementWhile(cond, stmt);
            nLoopCounter--;
        }
        )
    ;

ifstmt returns [PMLStatementIf r] throws Exception
{
    r = null;
    PMLCondition cond;
    PMLStatement thenPart;
    PMLStatement elsePart = null;
}
```

```
    : #("if"
        cond = pred:cond_expr
        {
            AST astThen = pred.getNextSibling();
            AST astElse = astThen.getNextSibling();
            thenPart = statement(astThen);
            if (astElse != null)
                elsePart = statement(astElse);
            r = new PMLStatementIf(cond, thenPart, elsePart);
        }
        )
    ;

returnstmt returns [PMLStatementReturn r] throws Exception
{
    r = null;
    PMLExpression pExpr = null;
}
    : #(RETURN_STMT
        {
            AST astExpr = #RETURN_STMT.getFirstChild();
            if (astExpr != null)
                pExpr = expr(astExpr);

            if (pCurrentFunction.m_return.getVariableType() == PMLVariable.typeVoid)
            {
                if (pExpr != null)
                    throw new Exception("Function '" + pCurrentFunction.getPrototype() +
"' should not return anything!");
            } else
            {
                if (pExpr == null)
                    throw new Exception("Function '" + pCurrentFunction.getPrototype() +
"' should return a value!");
            }

            r = new PMLStatementReturn(pExpr);
        }

        )
    ;

print_stmt returns [PMLStatementPrint r] throws Exception
{   r = null;
    Vector v = new Vector();
    PMLExpression pExpr;
    String s;
}
    : #("print"

        (
            pExpr = expr
                {
                    v.add(pExpr);
                }
            | s = string_const
              {
                v.add(s);
              }
        )+

        {
            r = new PMLStatementPrint(v);
        }

        )
    ;

string_const returns [String r] throws Exception
{ r = new String("Doh"); }
    : #(STRING_CONSTANT
        {
            r = new String(#STRING_CONSTANT.getText());
        }
        )
```

```
    ;

funccall_stmt returns [PMLStatementExpr r] throws Exception
{ r = null; PMLExpression pExpr = null; }
    : pExpr = funccall
        {
            r = new PMLStatementExpr(pExpr);
        }
    ;

argslist throws Exception
{ PMLVariable var; }
    : (
        var = arg
        {
            pCurrentFunction.addArgument(var);
        }
      )*
;

arg returns [PMLVariable r] throws Exception
{
    String var_type; r = null;
}
    : #(ID var_type=datatype
         {
            AST varName = #ID;
            r = PMLVariable.createVariable(var_type, varName.getText());
         }
      )
;

function throws Exception
    { String return_type;
      Vector args;
      PMLStatementBlock pStmtBlock;
    }
    : #("func" return_type=datatype ID
        {
            AST functionName = #ID;
            PMLFunction function = new PMLFunction();
            function.m_return = PMLVariable.createVariable(return_type, "returnVar");
            pSymbolTableStack.push(function);
            function.setParent(pcu);
            function.m_sFunctionName = functionName.getText();
            pCurrentFunction = function;
        }

        argslist
        {
            pcu.addFunction(function);
        }

        pStmtBlock=statement_block
        {
            function.m_statementBlock = pStmtBlock;
            pCurrentFunction = null;
            pSymbolTableStack.pop();
            if (!function.checkAllControlPathsForReturnValue(null))
                throw new Exception("Function: " + function.getPrototype() + "\nNot all
control paths return a value!");
        }

       )
    ;

datatype returns [String s]
    { s = ""; }
         : "int" { s = "int"; }
         | "float" { s = "float"; }
         | "poly" { s = "poly"; }
         | "char" { s = "char"; }
         | "term" { s = "term"; }
         | "termarray" { s = "termarray"; }
         | "poly_equ" { s = "poly_equ"; }
```

```
        | "void" { s = "void"; }
        | "chararray" { s = "chararray";}
        ;
```

B.2 JAVA CODE

PML.java (main class)

```
import java.io.*;
import antlr.CommonAST;

class PML {

    public static void compileFile(String fileName) throws Exception
    {
        PMLDebug.shez_println("PARSING <" + fileName + ">", 1);
        PML_LEXER lexer = new PML_LEXER(new FileInputStream(fileName));
        PMLParser parser = new PMLParser(lexer);
        parser.start();

        PMLDebug.shez_println("WALKING <" + fileName + ">", 1);
        PMLWalker walker = new PMLWalker();
        PMLCompilationUnit pcu = walker.begin((CommonAST)parser.getAST());
    }

    public static void main(String[] args)
    {
        try
        {
            if (args.length == 0)
            {
                System.out.println("Unable to compile. No files were passed.");
                System.out.println("Usage:\n\tjava PML (filename.src)*\n");
                return;
            }

            // create runtime environment
            new PMLRuntimeEnvironment();

            boolean compileOK = true;

            // compile files
            try
            {
                for (int i = 0; i < args.length; i++)
                {
                    PMLRuntimeEnvironment.getRE().m_sCurrentFile = args[i];
                    compileFile(args[i]);
                }
            } catch (Exception e)
            {
                compileOK = false;
                System.err.println("Error while compiling <" +
PMLRuntimeEnvironment.getRE().m_sCurrentFile + ">:\n" + e.getMessage());
            }


            if (compileOK)
            {
                try
                {
                    PMLDebug.shez_println("EXECUTING", 1);
                    PMLRuntimeEnvironment.getRE().Start();
                } catch (Exception e)
                {
                    if (PMLRuntimeEnvironment.getRE().getCurrentActivationRecord() ==
null)
                        throw e;

                    System.err.println("Runtime error!\nFile: "
```

```
                                        +
PMLRuntimeEnvironment.getRE().getCurrentActivationRecord().m_function.getPCU().m_sFileName
                                        + "\nFunction: "
                                        +
PMLRuntimeEnvironment.getRE().getCurrentActivationRecord().m_function.getPrototype()
                                        + "\nError Description:\n"
                                        + e.getMessage());
                }
            }
        } catch(Exception e)
        {
            System.err.println("Error: " + e.getMessage());
        }
    }
}
```

PMLRuntimeEnvironment.java

```
/*
 * PMLRuntimeEnvironment.java
 *
 * Created on November 12, 2003, 9:02 PM
 */

import java.util.*;

/**
 *
 * @author  Shezan
 */
public class PMLRuntimeEnvironment {

    public String m_sCurrentFile;

    // PML Compilation Units
    private Vector m_aPCUs;
    public void addPCU(PMLCompilationUnit pcu)
    {
        pcu.m_sFileName = m_sCurrentFile;
        m_aPCUs.add(pcu);
    }

    private static PMLRuntimeEnvironment oneAndOnly = null;
    public static PMLRuntimeEnvironment getRE()
    {
        return PMLRuntimeEnvironment.oneAndOnly;
    }

    /** Creates a new instance of PMLRuntimeEnvironment */
    public PMLRuntimeEnvironment() {
        PMLRuntimeEnvironment.oneAndOnly = this;
        m_aPCUs = new Vector();
        m_stackActivationRecords = new Stack();

        m_sCurrentFile = "stdlib.src";
        try
        {
            PMLCompilationUnit.createStdLibPCU();
        } catch (Exception e) { e.printStackTrace(); }
    }

    // stack of PMLFunctionInstance
    private Stack m_stackActivationRecords;

    public PMLFunctionInstance getCurrentActivationRecord()
    {
        if (m_stackActivationRecords.size() == 0)
            return null;
        else
            return (PMLFunctionInstance)m_stackActivationRecords.peek();
    }

    public PMLVariable findRuntimeVariable(String r_sSymbolName)
```

```
    {
        PMLFunctionInstance ar = getCurrentActivationRecord();
        PMLVariable ret = ar.findRuntimeVariable(r_sSymbolName);
        if (ret == null)
            ret = ar.m_function.getPCU().getVariable(r_sSymbolName);
        return ret;
    }

    private boolean m_bInitialised = false;
    public void Start() throws Exception
    {
        PMLFunction fnMain = null;

        for (int i = 0; i < m_aPCUs.size(); i++)
        {
            if (fnMain == null)
            {
                fnMain = ((PMLCompilationUnit)m_aPCUs.get(i)).Initialise();
            } else
            {
                ((PMLCompilationUnit)m_aPCUs.get(i)).Initialise();
            }
        }

        if (fnMain == null)
        {
            throw new Exception("'main' function without arguments not found!");
        }

        m_bInitialised = true;
        executeFunction(fnMain, new Vector());
    }

    public void findDuplicateFunction(PMLFunction function) throws Exception
    {
        if (function.m_sFunctionName.charAt(0) == '@')
            return;

        boolean isOK = true;

        PMLCompilationUnit pcu = null;
        int h,i;
        for (h = 0; h < m_aPCUs.size() && isOK; h++)
        {
            pcu = (PMLCompilationUnit)m_aPCUs.get(h);
            for (i = 0; i < pcu.m_aFunctions.size(); i++)
            {
                if (((PMLFunction)pcu.m_aFunctions.get(i)).isSimilar(function))
                {
                    isOK = false;
                    break;
                }
            }
        }

        if (isOK == false)
            throw new Exception("Function '" + function.m_sFunctionName + "' has already
been declared\nwith the same/similar arguments in <" + pcu.m_sFileName + ">!");
    }



    public PMLVariable executeFunction(String funcName, Vector params) throws Exception
    {
        PMLFunction func = null;
        PMLFunction fnTest;
        PMLVariable argTest;
        int h,i,j;
        PMLCompilationUnit pcu;
        for (h = 0; h < m_aPCUs.size() && func == null; h++)
        {
            pcu = (PMLCompilationUnit)m_aPCUs.get(h);
            for (i = 0; i < pcu.m_aFunctions.size(); i++)
            {
                fnTest = (PMLFunction)pcu.m_aFunctions.get(i);
```

```
                if (fnTest.m_sFunctionName.compareTo(funcName) == 0)
                {
                   if (fnTest.m_aArguments.size() == params.size())
                   {
                       try
                       {
                           for (j = 0; j < params.size(); j++)
                           {
                               argTest =
((PMLVariable)fnTest.m_aArguments.get(j)).getCopy();
                               argTest.setValue((PMLVariable)params.get(j));
                           }
                       } catch (Exception e)
                       {
                           continue;
                       }

                       func = fnTest;
                       break;
                   }
                }
            }
        }

        if (func == null)
        {
            String sParamList = Integer.toString(params.size()) + " params were
passed:\n";
            for (i = 0; i < params.size(); i++)
            {
                sParamList = sParamList + Integer.toString(i+1) + ". (" +
PMLVariable.types[((PMLVariable)params.get(i)).getVariableType()] + " " +
((PMLVariable)params.get(i)).getSymbolName() + ")\t-> " +
((PMLVariable)params.get(i)).getStringValue() + "\n";
            }
            sParamList = sParamList + "End of param list";
            throw new Exception("Invalid function call: '" + funcName + "'\nEither the
function doesn't exist or an invalid\nlist of parameters were passed.\n" + sParamList);
        }

        return executeFunction(func, params);
    }

    public PMLVariable executeFunction(PMLFunction func, Vector params) throws Exception
    {
        if (m_bInitialised == false
            && func.m_sFunctionName.charAt(0) != '@')
            throw new Exception("Not allowed to initialise global variables with function
calls!");

        if (params.size() != func.m_aArguments.size())
            throw new Exception("Function '" + func.m_sFunctionName + "' called with
invalid number of arguments!");

        int i;
        String depth = "";
        for (i = 0; i < m_stackActivationRecords.size(); i++)
            depth = depth + "   ";

        String funcProto = func.getPrototype();

        PMLDebug.shez_println(depth + "Running " + funcProto + "...", 3);

        Vector params_Copy = new Vector();
        PMLVariable paramVar;
        PMLVariable paramVar_Copy;
        PMLVariable argVar;
        for (i = 0; i < params.size(); i++)
        {
            argVar = (PMLVariable)func.m_aArguments.get(i);
            paramVar = (PMLVariable)params.get(i);
            paramVar_Copy = PMLVariable.createVariable(argVar.getVariableType(),
argVar.getSymbolName());
            paramVar_Copy.setValue(paramVar);
            params_Copy.add(paramVar_Copy);
```

```
        }

        // create the activation frame
        PMLFunctionInstance ar = new PMLFunctionInstance(func, params_Copy);
        m_stackActivationRecords.push(ar);
        try
        {
            ar.Run();
        } catch (PMLReturnException re) { }
        m_stackActivationRecords.pop();

        PMLDebug.shez_println(depth + "Returned from " + funcProto, 3);

        return ar.m_returnValue;
    }


}
```

PMLSymbolTable.java

```
/*
 * PMLSymbolTable.java
 *
 * Created on October 27, 2003, 5:38 PM
 */

/**
 *
 * @author  Shezan
 */
public interface PMLSymbolTable {
    public PMLVariable findVariable(String r_sSymbolName);
    public void addVariable(PMLVariable var) throws Exception;
    public void setParent(PMLSymbolTable parent);
}
```

PMLCompilationUnit.java

```
/*
 * PMLCompilationUnit.java
 *
 * Created on October 19, 2003, 2:28 PM
 */

import java.util.*;

/**
 *
 * @author  Shezan
 */
public class PMLCompilationUnit implements PMLSymbolTable {

    /** Creates a new instance of PMLCompilationUnit */
    public PMLCompilationUnit() {

        // add it to the current RE
        PMLRuntimeEnvironment.getRE().addPCU(this);

        m_aFunctions = new Vector();
        m_aVariables = new Vector();
    }

    public Vector m_aFunctions; // all functions
    public Vector m_aVariables; // global variables


    public static PMLCompilationUnit createStdLibPCU() throws Exception
    {
        PMLCompilationUnit ret = new PMLCompilationUnit();
        PMLFunction func;
```

```
    // coeff function
    func = new PMLFunction();
    func.setParent(ret);
    func.m_bBuiltIn = true;
    func.m_sFunctionName = "coeff";
    func.addArgument(new TermVariable("t"));
    ret.addFunction(func);

    // degree function
    func = new PMLFunction();
    func.setParent(ret);
    func.m_bBuiltIn = true;
    func.m_sFunctionName = "degree";
    func.addArgument(new TermVariable("t"));
    ret.addFunction(func);

    // degree function (with char argument)
    func = new PMLFunction();
    func.setParent(ret);
    func.m_bBuiltIn = true;
    func.m_sFunctionName = "degree";
    func.addArgument(new TermVariable("t"));
    func.addArgument(new CharVariable("c"));
    ret.addFunction(func);

    // length
    func = new PMLFunction();
    func.setParent(ret);
    func.m_bBuiltIn = true;
    func.m_sFunctionName = "length";
    func.addArgument(new TermArray("ta"));
    ret.addFunction(func);

    // length
    func = new PMLFunction();
    func.setParent(ret);
    func.m_bBuiltIn = true;
    func.m_sFunctionName = "length";
    func.addArgument(new CharArray("ca"));
    ret.addFunction(func);

    // polyterm
    func = new PMLFunction();
    func.setParent(ret);
    func.m_bBuiltIn = true;
    func.m_sFunctionName = "polyterm";
    func.addArgument(new PolyVariable("p"));
    ret.addFunction(func);

    /*
    // variable
    func = new PMLFunction();
    func.setParent(ret);
    func.m_bBuiltIn = true;
    func.m_sFunctionName = "variable";
    func.addArgument(new TermVariable("t"));
    ret.addFunction(func);
     */

    // variable
    func = new PMLFunction();
    func.setParent(ret);
    func.m_bBuiltIn = true;
    func.m_sFunctionName = "variable";
    func.addArgument(new PolyVariable("pa"));
    ret.addFunction(func);
    return ret;
}

// initialise all global variables in the pcu
// will return a pointer to the main function
// if it is in this pcu
public PMLFunction Initialise() throws Exception
{
    PMLFunction fnMain = null;
```

```
        PMLFunction fnTest;
        for (int i = 0; i < m_aFunctions.size(); i++)
        {
            fnTest = (PMLFunction)m_aFunctions.get(i);
            if (fnTest.m_sFunctionName.compareTo("main") == 0
                && fnTest.m_aArguments.size() == 0)
            {
                fnMain = fnTest;
            } else if (fnTest.m_sFunctionName.charAt(0) == '@')
            {
                PMLRuntimeEnvironment.getRE().executeFunction(fnTest, new Vector());
            }
        }

        return fnMain;
    }

    public PMLVariable getVariable(String r_sSymbolName)
    {
        int i;
        PMLVariable var;
        for (i = 0; i < m_aVariables.size(); i++)
        {
            var = (PMLVariable)m_aVariables.get(i);
            if (var.getSymbolName().compareTo(r_sSymbolName) == 0)
                return var;
        }
        return null;
    }

    public void addVariable(PMLVariable variable) throws Exception
    {
        if (findVariable(variable.getSymbolName()) != null)
            throw new Exception("Variable '" + variable.getSymbolName() + "' has already
been declared!");
        else
            m_aVariables.add(variable);
    }


    public void addFunction(PMLFunction function) throws Exception
    {
        PMLRuntimeEnvironment.getRE().findDuplicateFunction(function);
        m_aFunctions.add(function);
    }

    public PMLVariable findVariable(String r_sSymbolName) {
        return getVariable(r_sSymbolName);
    }

    public void setParent(PMLSymbolTable parent) {
    }

    public String m_sFileName;
}
```

PMLVariable.java

```
/*
 * PMLVariable.java
 *
 * Created on October 19, 2003, 1:52 PM
 */
/*
 * modified termvariable class by making LetterDegree a separate class
 * Termarray class added
 * Hari Kurup - 09-Nov-2003
*/

/*
 * Modified by Shezan - 12 Nov '03
 * Added CharVariable class
 */
```

```
/*
 * Modified by Shezan - 19 Nov '03
 * Added negate() and compare() functions
 */
/*
 * Modified by Hari Kurup - 27-Nov-2003
 * Added chararray class
 */
/*
 * Modified by Hari Kurup - 01-Dec-2003
 * CharArray will print contiguous elements without separation
 */
import java.util.*;

/**
 *
 * @author  Shezan
 */
public abstract class PMLVariable {

    public static final int typeVoid = 0;
    public static final int typeInt = 1;
    public static final int typeFloat = 2;
    public static final int typeTerm = 3;
    public static final int typePoly = 4;
    public static final int typeTermArray = 5;
    public static final int typeChar = 6;
    public static final int typeCharArray = 7;
    public static final String types[] = { "void", "int", "float", "term", "poly",
"termarray", "char","chararray" };

    public static String removeDot0(String s)
    {
        if (s.substring(s.length()-2).compareTo(".0") == 0)
            return s.substring(0, s.length()-2);
        else
            return s;
    }

    public abstract boolean compare(PMLVariable var) throws Exception;
    public abstract PMLVariable negate() throws Exception;

    public PMLVariable getCopy()
    {
        try
        {
            PMLVariable c = PMLVariable.createVariable(getVariableType(),
getSymbolName());
            c.setValue(this);
            return c;
        } catch (PMLNoValidConversionException e) {
        } catch (Exception e) {
        }
        System.err.println("ERROR COPYING VARIABLES");
        return null;
    }

    private String m_sSymbolName;

    /** Creates a new instance of PMLVariable */
    public PMLVariable(String r_sSymbolName) {
        m_sSymbolName = r_sSymbolName;
    }

    public String getSymbolName() { return m_sSymbolName; }


    public abstract int getVariableType();
    public void setValue(PMLVariable r) throws PMLNoValidConversionException, Exception
    {
        setValue(r, false);
    }
    public abstract void setValue(PMLVariable r, boolean force) throws
PMLNoValidConversionException, Exception;
```

```
    public abstract String getStringValue() throws Exception;
    public abstract void setValue(String s) throws Exception;


    public static PMLVariable createVariable(int r_nType, String r_sSymbolName) throws
Exception
    {
       // System.out.println(" symbol = " + r_sSymbolName + " type = " + r_nType);
        if (r_nType >= 0 && r_nType < PMLVariable.types.length)
            return createVariable(PMLVariable.types[r_nType], r_sSymbolName);
        else
            throw new Exception("Invalid variable type: '" + Integer.toString(r_nType) +
"'");
    }

    public static PMLVariable createVariable(String r_sType, String r_sSymbolName) throws
Exception
    {
        if (r_sType.compareTo("int") == 0)
            return new IntVariable(r_sSymbolName);
        else if (r_sType.compareTo("float") == 0)
            return new FloatVariable(r_sSymbolName);
        else if (r_sType.compareTo("term") == 0)
            return new TermVariable(r_sSymbolName);
        else if (r_sType.compareTo("poly") == 0)
            return new PolyVariable(r_sSymbolName);
        else if (r_sType.compareTo("void") == 0)
            return new VoidVariable(r_sSymbolName);
        else if (r_sType.compareTo("termarray") == 0)
            return new TermArray(r_sSymbolName);
        else if (r_sType.compareTo("char") == 0)
            return new CharVariable(r_sSymbolName);
        else if (r_sType.compareTo("chararray") == 0)
            return new CharArray(r_sSymbolName);
        else
            throw new Exception("Invalid variable type: '" + r_sType + "'");
    }
}

class VoidVariable extends PMLVariable
{
    public VoidVariable(String r_sSymbolName)
    {
        super(r_sSymbolName);
    }

    public String getStringValue() throws Exception {
        return "<<VOID>>";
    }

    public int getVariableType() {
        return PMLVariable.typeVoid;
    }

    public void setValue(PMLVariable r, boolean force) throws
PMLNoValidConversionException, Exception {
        if (r.getVariableType() != PMLVariable.typeVoid)
            throw new PMLNoValidConversionException(this, r);
    }

    public void setValue(String s) throws Exception {
        throw new Exception("Unable to set value for void variable!");
    }

    public boolean compare(PMLVariable var) throws Exception {
        throw new Exception("Cannot compare void expression!");
    }

    public PMLVariable negate() throws Exception {
        throw new Exception("Cannot negate void expression!");
    }

}

class CharVariable extends PMLVariable
```

```
{
    public boolean m_bPositive;
    public char m_cValue;

    public CharVariable(String r_sSymbolName)
    {
        super(r_sSymbolName);
        m_bPositive = true;
    }

    public String getStringValue() throws Exception {
        char doh[] = { m_cValue };
        if (m_bPositive)
            return new String(doh);
        else
            return "-" + new String(doh);
    }

    public int getVariableType() {
        return PMLVariable.typeChar;
    }

    public void setValue(PMLVariable r, boolean force) throws
PMLNoValidConversionException, Exception {
        if (r.getVariableType() == PMLVariable.typeChar)
        {
            CharVariable cr = (CharVariable)r;
            m_cValue = cr.m_cValue;
            m_bPositive = cr.m_bPositive;
        } else
            throw new PMLNoValidConversionException(this, r);
    }

    public void setValue(String s) throws Exception {
    }

    public boolean compare(PMLVariable var) throws Exception {
        if (var.getVariableType() == PMLVariable.typeChar)
        {
            CharVariable cv = (CharVariable)var;
            return (cv.m_bPositive == m_bPositive
                    && cv.m_cValue == m_cValue);
        } else if (var.getVariableType() == PMLVariable.typeTerm
                    || var.getVariableType() == PMLVariable.typePoly)
        {
            return var.compare(this);
        } else
            return false;
    }

    public PMLVariable negate() throws Exception {
        CharVariable var = (CharVariable)getCopy();
        var.m_bPositive = !var.m_bPositive;
        return var;
    }

}

/*CharArray datatype -- HK */
class CharArray extends PMLVariable
{
    public Vector m_aChars;

    public CharArray(String r_sSymbolName)
    {
        super(r_sSymbolName);
        m_aChars = new Vector();
    }

    public CharVariable getTerm(int intIndex) throws Exception
    {
        return (CharVariable)m_aChars.get(intIndex);
    }

    public String getStringValue( int intIndex ) throws Exception
```

```
    {
        String ret = "";
        int i;
        CharVariable chrVal;
        if (( intIndex-1 < m_aChars.size() ) && ( intIndex -1 >= 0 ))
        {
            chrVal = (CharVariable)m_aChars.get(intIndex-1);
            return chrVal.getStringValue();
        } else
            return "<<Invalid Index>>";
    }

     /* - CharArray will not separate each term with a comma -- HK */
    public String getStringValue() throws Exception {
        String ret = "";
        int i;
        CharVariable chrVal;
        String sChar;
        for (i = 0; i < m_aChars.size(); i++)
        {
            chrVal = (CharVariable)m_aChars.get(i);
            sChar = chrVal.getStringValue();

            if (ret.length() != 0)
                sChar = ", " + sChar;

            ret += sChar;
        }

        if (ret.length() == 0)
            ret = "<<Empty Array>>";
        return ret;
    }

    public int getVariableType() {
        return PMLVariable.typeCharArray;
    }

    /* CharArray can only be assigned another CharArray  --  HK*/
    public void setValue(PMLVariable r, boolean force) throws
PMLNoValidConversionException, Exception {
        if (r.getVariableType() == PMLVariable.typeCharArray)
        {
            CharArray ca = (CharArray)r;
            m_aChars = new Vector();
            for (int i = 0; i < ca.m_aChars.size(); i++)
                m_aChars.add(((PMLVariable)ca.m_aChars.get(i)).getCopy());
        } else
            throw new PMLNoValidConversionException(this, r);
    }

    public void setValue(String s) throws Exception {
    }

    public boolean compare(PMLVariable var) throws Exception {
        throw new Exception("Cannot compare 'Chararray' object!");
    }

    public PMLVariable negate() throws Exception {
        throw new Exception("Cannot negate 'Chararray' object!");
    }

}


class IntVariable extends PMLVariable
{
    public int m_nValue;

    public IntVariable(String r_sSymbolName)
    {
        super(r_sSymbolName);
        m_nValue = 0;
    }
```

```
    public int getVariableType() {
        return PMLVariable.typeInt;
    }

    public String getStringValue() throws Exception
    {
        return String.valueOf(m_nValue);
    }

    public void setValue(PMLVariable r, boolean force) throws
PMLNoValidConversionException, Exception {
        if (r.getVariableType() == PMLVariable.typeFloat)
        {
            FloatVariable r2 = (FloatVariable)r;
            if (force)
            {
                m_nValue = (int)r2.m_fValue;
            } else
            {
                String sVal = r.getStringValue();
                if (sVal.substring(sVal.length()-2).compareTo(".0") == 0)
                {
                    m_nValue = (int)r2.m_fValue;
                } else
                {
                    throw new Exception("Unable to demote '" + sVal + "' to int!");
                }
            }
        } else if (r.getVariableType() == PMLVariable.typeInt)
        {
            IntVariable r2 = (IntVariable)r;
            m_nValue = r2.m_nValue;
        } else if (force && (r.getVariableType() == PMLVariable.typeTerm
                             || r.getVariableType() == PMLVariable.typePoly))
        {
            // lazy
            FloatVariable fv = new FloatVariable("temp");
            fv.setValue(r, true);
            setValue(fv, true);
        } else
            throw new PMLNoValidConversionException(this, r);
    }

    public void setValue(String s) throws Exception {
        m_nValue = Integer.parseInt(s);
    }

    public boolean compare(PMLVariable var) throws Exception {
        if (var.getVariableType() == PMLVariable.typeInt)
        {
            IntVariable iv = (IntVariable)var;
            return (iv.m_nValue == m_nValue);
        } else if (var.getVariableType() == PMLVariable.typeFloat)
        {
            FloatVariable fv = (FloatVariable)var;
            return (fv.m_fValue == (float)m_nValue);
        } else if (var.getVariableType() == PMLVariable.typeTerm
                    || var.getVariableType() == PMLVariable.typePoly)
        {
            return var.compare(this);
        } else
            return false;
    }

    public PMLVariable negate() throws Exception {
        IntVariable var = (IntVariable)getCopy();
        var.m_nValue = -var.m_nValue;
        return var;
    }

}

class FloatVariable extends PMLVariable
{
    public float m_fValue;
```

```
    public int getVariableType() {
        return PMLVariable.typeFloat;
    }

    public String getStringValue() throws Exception
    {
        PMLDebug.println("=====the correct one===");
        return PMLVariable.removeDot0(String.valueOf(m_fValue));
    }

    public void setValue(String s) throws Exception
    {
        m_fValue = Float.parseFloat(s);
    }
    public void setValue(PMLVariable r, boolean force) throws
PMLNoValidConversionException, Exception {
        if (r.getVariableType() == PMLVariable.typeFloat)
        {
            FloatVariable r2 = (FloatVariable)r;
            m_fValue = r2.m_fValue;
        } else if (r.getVariableType() == PMLVariable.typeInt)
        {
            IntVariable r2 = (IntVariable)r;
            m_fValue = r2.m_nValue;
        } else if (force && r.getVariableType() == PMLVariable.typeTerm)
        {
            TermVariable r2 = (TermVariable)r;
            if (r2.m_aLetterDegrees.size() > 0)
                throw new Exception("Unable to demote '" + r2.getStringValue() + "' to
int/float!");
            else
                setValue(r2.m_fCoeff.evaluate());
        } else if (force && r.getVariableType() == PMLVariable.typePoly)
        {
            TermVariable tv = new TermVariable("temp");
            tv.setValue(r, true);
            setValue(tv, true);
        } else
            throw new PMLNoValidConversionException(this, r);
    }

    public boolean compare(PMLVariable var) throws Exception {
        if (var.getVariableType() == PMLVariable.typeInt)
        {
            IntVariable iv = (IntVariable)var;
            return ((float)iv.m_nValue == m_fValue);
        } else if (var.getVariableType() == PMLVariable.typeFloat)
        {
            FloatVariable fv = (FloatVariable)var;
            return (fv.m_fValue == m_fValue);
        } else if (var.getVariableType() == PMLVariable.typeTerm
                   || var.getVariableType() == PMLVariable.typePoly)
        {
            return var.compare(this);
        } else
            return false;
    }

    public PMLVariable negate() throws Exception {
        FloatVariable var = (FloatVariable)getCopy();
        var.m_fValue = -var.m_fValue;
        return var;
    }

    public FloatVariable(String r_sSymbolName)
    {
        super(r_sSymbolName);
        m_fValue = 0.0f;
    }
}
 class LetterDegree
    {
        public LetterDegree() { }
        public char m_cLetter;
```

```
        public PMLExpression m_fDegree;
    // constructor that sets the variable and degree -- HK
        public LetterDegree( char c, PMLExpression f )
        {
            m_cLetter = c;
            m_fDegree = f;
        }
        public LetterDegree getCopy() throws Exception
        {
            LetterDegree n = new LetterDegree();
            n.m_cLetter = m_cLetter;
            n.m_fDegree = new PMLExprLiteral(m_fDegree.evaluate().getCopy());
            return n;
        }

        public boolean compare(LetterDegree ld) throws Exception
        {
            if (ld.m_cLetter == m_cLetter)
                return m_fDegree.evaluate().compare(ld.m_fDegree.evaluate());
            else
                return false;
        }
    }


class TermVariable extends PMLVariable
{
    public PMLExpression m_fCoeff;
    public Vector m_aLetterDegrees;

   //  CreateLetter will create the variable part of the term -- HK
    public  LetterDegree CreateLetter ( char c, PMLExpression f )
    {
        LetterDegree letDegree = new LetterDegree(c, f);
        return letDegree;
    }

    public TermVariable(String r_sSymbolName)
    {
        super(r_sSymbolName);
        m_aLetterDegrees = new Vector();
        IntVariable var0 = new IntVariable("tmp");
        var0.m_nValue = 0;
        m_fCoeff = new PMLExprLiteral(var0);
    }

    // returns true for numeric & lower case chars only
    public static boolean isComplexString(String s)
    {
        if (s.length() == 0)
            return false;
        int i;
        boolean isChar = (s.charAt(0) >= 'a' && s.charAt(0) <= 'z')
                            || (s.length() > 1 && (s.charAt(0) == '-' || s.charAt(0) ==
'+') && s.charAt(1) >= 'a' && s.charAt(1) <= 'z');
        boolean isNum = ((s.charAt(0) >= '0' && s.charAt(0) <= '9') || s.charAt(0) == '.')
                            || (s.length() > 1 && (s.charAt(0) == '-' || s.charAt(0) ==
'+') && ((s.charAt(1) >= '0' && s.charAt(1) <= '9') || s.charAt(1) == '.'));
        if (!isChar && !isNum)
            return true;

        for (i = 1; i < s.length(); i++)
        {
            if (isChar)
            {
                if (s.charAt(i) < 'a' || s.charAt(i) > 'z')
                    return true;
            } else if (isNum)
            {
                if ((s.charAt(i) < '0' || s.charAt(i) > '9') && s.charAt(i) != '.')
                    return true;
            }
        }
        return false;
    }
```

```
    public String getStringValue() throws Exception {
        String ret = m_fCoeff.evaluate().getStringValue();
        if (ret.compareTo("1") == 0 && m_aLetterDegrees.size() > 0)
            ret = "";
        else if (ret.compareTo("-1") == 0 && m_aLetterDegrees.size() > 0)
            ret = "-";
        else if (isComplexString(ret))
            ret = "(" + ret + ")";

        LetterDegree ld;
        String degree;
        int i;
        for (i = 0; i < m_aLetterDegrees.size(); i++)
        {
            ld = (LetterDegree)m_aLetterDegrees.get(i);
            degree = ld.m_fDegree.evaluate().getStringValue();
            if (degree.compareTo("0") == 0)
                continue;

            char doh[] = { ld.m_cLetter };
            ret += new String(doh);
            if (!isComplexString(degree))
            {
                if (degree.compareTo("1") != 0)
                    ret += "^" + degree;
            } else
            {
                ret += "^(" + degree + ")";
            }
        }

        return ret;
    }

    public void setValue(PMLVariable r, boolean force) throws
PMLNoValidConversionException, Exception {
        if (r.getVariableType() == PMLVariable.typeChar)
        {
            CharVariable cv = (CharVariable)r;
            IntVariable var1 = new IntVariable("tmp");
            if (cv.m_bPositive == false)
                var1.m_nValue = -1;
            else
                var1.m_nValue = 1;
            m_fCoeff = new PMLExprLiteral(var1);
            m_aLetterDegrees = new Vector();
            m_aLetterDegrees.add(CreateLetter(cv.m_cValue, new PMLExprLiteral(var1)));
        } else
        if (r.getVariableType() == PMLVariable.typeInt
            || r.getVariableType() == PMLVariable.typeFloat/*
            || r.getVariableType() == PMLVariable.typeChar*/)
        {
            m_fCoeff = new PMLExprLiteral(r);
            m_aLetterDegrees = new Vector();
        } else if (r.getVariableType() == PMLVariable.typeTerm)
        {
            TermVariable tv = (TermVariable)r;
            m_fCoeff = new PMLExprLiteral(tv.m_fCoeff.evaluate().getCopy());
            m_aLetterDegrees = new Vector();
            int i;
            for (i = 0; i < tv.m_aLetterDegrees.size(); i++)

m_aLetterDegrees.add(((LetterDegree)tv.m_aLetterDegrees.get(i)).getCopy());
        } else if (force && r.getVariableType() == PMLVariable.typePoly)
        {
            PolyVariable pv = (PolyVariable)r;
            if (pv.m_aTerms.size() == 1)
                setValue((TermVariable)pv.m_aTerms.get(0));
            else if (pv.m_aTerms.size() == 0)
                setValue(PMLMath.CreateZeroTerm());
            else
                throw new Exception("Unable to demote '" + pv.getStringValue() + "' to
term/float/int!");
        } else
```

```
            throw new PMLNoValidConversionException(this, r);
    }

    public int getVariableType() {
        return PMLVariable.typeTerm;
    }

    public void setValue(String s) throws Exception {
    }

    public boolean compare(PMLVariable var) throws Exception {
        if (var.getVariableType() == PMLVariable.typeTerm)
        {
            TermVariable tv = (TermVariable)var;
            if (m_fCoeff.evaluate().compare(tv.m_fCoeff.evaluate()))
            {
                if (tv.m_aLetterDegrees.size() == m_aLetterDegrees.size())
                {
                    int i,j;
                    LetterDegree ld1, ld2;
                    boolean checked[] = new boolean[tv.m_aLetterDegrees.size()];
                    for (i = 0; i < tv.m_aLetterDegrees.size(); i++)
                        checked[i] = false;

                    for (i = 0; i < tv.m_aLetterDegrees.size(); i++)
                    {
                        ld1 = (LetterDegree)tv.m_aLetterDegrees.get(i);
                        for (j = 0; j < m_aLetterDegrees.size(); j++)
                        {
                            if (checked[j])
                                continue;

                            ld2 = (LetterDegree)m_aLetterDegrees.get(j);
                            if (ld1.compare(ld2))
                            {
                                checked[j] = true;
                                break;
                            }
                        }
                        if (j == m_aLetterDegrees.size())
                            return false;
                    }

                    return true;
                } else
                    return false;
            } else
                return false;
        } else if (var.getVariableType() == PMLVariable.typeChar)
        {
            if (m_aLetterDegrees.size() == 1)
            {
                CharVariable cv = (CharVariable)var;
                LetterDegree ld = (LetterDegree)m_aLetterDegrees.get(0);
                if (ld.m_cLetter == cv.m_cValue)
                {
                    PMLVariable coeff = m_fCoeff.evaluate();
                    FloatVariable fcoeff = new FloatVariable("tmp");
                    try
                    {
                        fcoeff.setValue(coeff, true);
                    } catch (Exception e)
                    {
                        return false;
                    }

                    if ((cv.m_bPositive && fcoeff.getStringValue().compareTo("1") == 0)
                        || (!cv.m_bPositive && fcoeff.getStringValue().compareTo("-1") ==
0))
                    {
                        PMLVariable deg = ld.m_fDegree.evaluate();
                        FloatVariable fdeg = new FloatVariable("tmp");
                        try
                        {
                            fdeg.setValue(deg, true);
```

```
                        } catch (Exception e)
                        {
                            return false;
                        }

                        return (fdeg.getStringValue().compareTo("1") == 0);
                    } else
                        return false;
                } else
                    return false;
            } else
                return false;
        } else if (var.getVariableType() == PMLVariable.typeInt
                    || var.getVariableType() == PMLVariable.typeFloat/*
                    || var.getVariableType() == PMLVariable.typeChar*/)
        {
            if (m_aLetterDegrees.size() == 0)
                return (m_fCoeff.evaluate().compare(var));
            else
                return false;
        } else if (var.getVariableType() == PMLVariable.typePoly)
        {
            return var.compare(this);
        } else
            return false;
    }

    public PMLVariable negate() throws Exception {
        TermVariable var = (TermVariable)getCopy();
        var.m_fCoeff = new PMLExprLiteral(var.m_fCoeff.evaluate().negate());
        return var;
    }

}

//  Termarray class for termarray variables -- HK
class TermArray extends PMLVariable
{
    public Vector m_aTerms;

    public TermArray(String r_sSymbolName)
    {
        super(r_sSymbolName);
        m_aTerms = new Vector();
    }

    public TermVariable getTerm(int intIndex) throws Exception
    {
        return (TermVariable)m_aTerms.get(intIndex);
    }

    public String getStringValue( int intIndex ) throws Exception
    {
        String ret = "";
        int i;
        TermVariable term;
        if (( intIndex-1 < m_aTerms.size() ) && ( intIndex -1 >= 0 ))
        {
            term = (TermVariable)m_aTerms.get(intIndex-1);
            ret = term.getStringValue();
        }
        return ret;
    }

    /* modified by shezan
     * - TermArray will separate each term with a comma instead of '+'
     *   because '+' looks like a poly
     */
    public String getStringValue() throws Exception {
        String ret = "";
        int i;
        TermVariable term;
        String sTerm;
        for (i = 0; i < m_aTerms.size(); i++)
        {
```

```
            term = (TermVariable)m_aTerms.get(i);
            sTerm = term.getStringValue();
            if (ret.length() != 0)
                sTerm = ", " + sTerm;

            ret += sTerm;
        }

        if (ret.length() == 0)
            ret = "<<Empty Array>>";

        return ret;
    }

    public int getVariableType() {
        return PMLVariable.typeTermArray;
    }

    /* modified by shezan. TermArray can only be assigned another TermArray */
    public void setValue(PMLVariable r, boolean force) throws
PMLNoValidConversionException, Exception {
        if (r.getVariableType() == PMLVariable.typeTermArray)
        {
            TermArray ta = (TermArray)r;
            m_aTerms = new Vector();
            for (int i = 0; i < ta.m_aTerms.size(); i++)
                m_aTerms.add(((PMLVariable)ta.m_aTerms.get(i)).getCopy());
        } else
            throw new PMLNoValidConversionException(this, r);
    }

    public void setValue(String s) throws Exception {
    }

    public boolean compare(PMLVariable var) throws Exception {
        throw new Exception("Cannot compare 'termarray' object!");
    }

    public PMLVariable negate() throws Exception {
        throw new Exception("Cannot negate 'termarray' object!");
    }

}



class PolyVariable extends PMLVariable
{
    public Vector m_aTerms;

    public PolyVariable(String r_sSymbolName)
    {
        super(r_sSymbolName);
        m_aTerms = new Vector();
    }

    public String getStringValue() throws Exception {
        String ret = "";
        int i;
        TermVariable term;
        String sTerm;
        for (i = 0; i < m_aTerms.size(); i++)
        {
            term = (TermVariable)m_aTerms.get(i);
            sTerm = term.getStringValue();
            if (sTerm.length() > 0)
            {
                if (ret.length() != 0)
                    if (sTerm.charAt(0) != '-')
                        sTerm = " + " + sTerm;
                    else
                        sTerm = " - " + sTerm.substring(1);

                ret += sTerm;
```

```
            }
        }

        if (ret.length() == 0)
            ret = "0";

        return ret;
    }

    public int getVariableType() {
        return PMLVariable.typePoly;
    }

    public void setValue(PMLVariable r, boolean force) throws
PMLNoValidConversionException, Exception {
        if (r.getVariableType() == PMLVariable.typeChar
            || r.getVariableType() == PMLVariable.typeInt
            || r.getVariableType() == PMLVariable.typeFloat)
        {
            // lazy
            TermVariable tv = new TermVariable("temp");
            tv.setValue(r);
            setValue(tv);
        } else if (r.getVariableType() == PMLVariable.typeTerm)
        {
            TermVariable tv = (TermVariable)r;
            m_aTerms = new Vector();
            m_aTerms.add(tv.getCopy());
        } else if (r.getVariableType() == PMLVariable.typePoly)
        {
            PolyVariable pv = (PolyVariable)r;
            m_aTerms = new Vector();
            int i;
            for (i = 0; i < pv.m_aTerms.size(); i++)
                m_aTerms.add(((TermVariable)pv.m_aTerms.get(i)).getCopy());
        } else
            throw new PMLNoValidConversionException(this, r);
    }

    public void setValue(String s) throws Exception {
    }

    public boolean compare(PMLVariable var) throws Exception {
        if (var.getVariableType() == PMLVariable.typePoly)
        {
            PolyVariable pv = (PolyVariable)var;
            if (pv.m_aTerms.size() == m_aTerms.size())
            {
                int i,j;
                TermVariable term1, term2;
                boolean checked[] = new boolean[pv.m_aTerms.size()];
                for (i = 0; i < pv.m_aTerms.size(); i++)
                    checked[i] = false;

                for (i = 0; i < pv.m_aTerms.size(); i++)
                {
                    term1 = (TermVariable)pv.m_aTerms.get(i);
                    for (j = 0; j < m_aTerms.size(); j++)
                    {
                        if (checked[j])
                            continue;
                        term2 = (TermVariable)m_aTerms.get(j);
                        if (term2.compare(term1))
                        {
                            checked[j] = true;
                            break;
                        }
                    }
                    if (j == m_aTerms.size())
                        return false;
                }

                return true;
            } else
                return false;
```

```
        } else if (var.getVariableType() == PMLVariable.typeChar
                    || var.getVariableType() == PMLVariable.typeInt
                    || var.getVariableType() == PMLVariable.typeFloat
                    || var.getVariableType() == PMLVariable.typeTerm)
        {
            if (m_aTerms.size() == 1)
            {
                // lazy
                PolyVariable pv = new PolyVariable("tmp");
                pv.setValue(var);
                return compare(pv);
            } else
                return false;
        } else
            return false;
    }

    public PMLVariable negate() throws Exception {
        PolyVariable var = (PolyVariable)getCopy();
        int i;
        for (i = 0; i < var.m_aTerms.size(); i++)
            var.m_aTerms.set(i, ((TermVariable)var.m_aTerms.get(i)).negate());
        return var;
    }

}
```

PMLFunction.java

```
import java.util.*;

public class PMLFunction implements PMLSymbolTable
{
    public PMLFunction()
    {
        try {
            m_return = PMLVariable.createVariable(PMLVariable.typeVoid, "returnVar");
        } catch (Exception e) { }
        m_aArguments = new Vector();
        m_bBuiltIn = false;
    }


    public boolean checkAllControlPathsForReturnValue(PMLStatement stmt) throws Exception
    {
        if (m_return.getVariableType() == PMLVariable.typeVoid)
            return true;

        if (stmt == null)
        {
            return checkAllControlPathsForReturnValue(m_statementBlock);
        }

        if (stmt.getClass().getName().compareTo("PMLStatementIf") == 0)
        {
            PMLStatementIf ifstmt = (PMLStatementIf)stmt;
            if (ifstmt.getElsePart() == null)
                return false;
            else
                return (checkAllControlPathsForReturnValue(ifstmt.getThenPart())
                        && checkAllControlPathsForReturnValue(ifstmt.getElsePart()));
        } else if (stmt.getClass().getName().compareTo("PMLStatementBlock") == 0)
        {
            PMLStatementBlock stmtBlock = (PMLStatementBlock)stmt;
            for (int i = 0; i < stmtBlock.getStatements().size(); i++)
            {
                if
(checkAllControlPathsForReturnValue((PMLStatement)stmtBlock.getStatements().get(i)))
                    return true;
            }

            return false;
        } else if (stmt.getClass().getName().compareTo("PMLStatementReturn") == 0)
```

```
        {
            return true;
        }

        return false;
    }

    public String getPrototype()
    {
        String funcProto = "";
        int i;
        for (i = 0; i < m_aArguments.size(); i++)
        {
            if (funcProto.length() == 0)
                funcProto =
PMLVariable.types[((PMLVariable)m_aArguments.get(i)).getVariableType()];
            else
                funcProto = funcProto + ", " +
PMLVariable.types[((PMLVariable)m_aArguments.get(i)).getVariableType()];
        }
        funcProto = m_sFunctionName + "(" + funcProto + ")";

        return funcProto;
    }

    public boolean isSimilar(PMLFunction f)
    {
        if (f.m_sFunctionName.compareTo(m_sFunctionName) == 0)
        {
            if (f.m_aArguments.size() == m_aArguments.size())
            {
                PMLVariable v1, v2;
                for (int i = 0; i < m_aArguments.size(); i++)
                {
                    v1 = (PMLVariable)f.m_aArguments.get(i);
                    v2 = (PMLVariable)m_aArguments.get(i);

                    if (v1.getVariableType() == v2.getVariableType())
                        continue;
                    if (v1.getVariableType() == PMLVariable.typeInt
                        && v2.getVariableType() == PMLVariable.typeFloat)
                        continue;
                    if (v1.getVariableType() == PMLVariable.typeFloat
                        && v2.getVariableType() == PMLVariable.typeInt)
                        continue;

                    return false;
                }
                return true;
            } else
                return false;
        } else
            return false;
    }

    public void addArgument(PMLVariable arg) throws Exception
    {
        if (findVariable(arg.getSymbolName()) == null)
        {
            m_aArguments.add(arg);
        } else
        {
            throw new Exception("Error: argument '" + arg.getSymbolName() + "' already
defined!");
        }
    }

    public void addVariable(PMLVariable var) throws Exception {
        addArgument(var);
    }

    public PMLVariable findVariable(String r_sSymbolName) {
        for (int i = 0; i < m_aArguments.size(); i++)
        {
            if
```

```
(((PMLVariable)m_aArguments.get(i)).getSymbolName().compareTo(r_sSymbolName) == 0)
                return (PMLVariable)m_aArguments.get(i);
        }

        return m_pSymbolTableParent.findVariable(r_sSymbolName);
    }

    private PMLSymbolTable m_pSymbolTableParent;
    public void setParent(PMLSymbolTable parent) {
        m_pSymbolTableParent = parent;
    }
    public PMLCompilationUnit getPCU() {
        return (PMLCompilationUnit)m_pSymbolTableParent;
    }

    public boolean m_bBuiltIn;
    public PMLVariable m_return;
    public String m_sFunctionName;
    public Vector m_aArguments;
    public PMLStatementBlock m_statementBlock;
}
```

PMLFunctionInstance.java

```
/*
 * PMLFunctionInstance.java
 *
 * Created on October 19, 2003, 2:05 PM
 */

import java.util.*;

class RuntimeVariable
{
    public RuntimeVariable() { }

    public PMLVariable m_pVar;
    public RuntimeVariable m_pNext;
}

/**
 * This contains the "Activation Record"
 * @author  Shezan
 */
public class PMLFunctionInstance {

    /** Creates a new instance of PMLFunctionInstance */
    public PMLFunctionInstance(PMLFunction fn, Vector params) {
        m_function = fn;
        m_runtimeVars = null;

        if (fn.m_return != null)
            m_returnValue = fn.m_return.getCopy();

        int i;
        for (i = 0; i < params.size(); i++)
            addRuntimeVariable((PMLVariable)params.get(i));
    }

    private RuntimeVariable m_runtimeVars;
    public void addRuntimeVariable(PMLVariable var)
    {
        RuntimeVariable rv = new RuntimeVariable();
        rv.m_pVar = var;
        rv.m_pNext = m_runtimeVars;
        m_runtimeVars = rv;
    }
    public void removeRuntimeVariable(int nRemoveCount)
    {
        while (nRemoveCount > 0 && m_runtimeVars != null)
        {
            m_runtimeVars = m_runtimeVars.m_pNext;
            nRemoveCount--;
```

```
    }
}
public PMLVariable findRuntimeVariable(String r_sSymbolName)
{
    RuntimeVariable rv = m_runtimeVars;
    while (rv != null)
    {
      // System.out.println(" symbol name = " + r_sSymbolName);
       if (rv.m_pVar.getSymbolName().compareTo(r_sSymbolName) == 0)
       {
       //    System.out.println(" ===");
   //      System.out.println("+++" + rv.m_pVar.getSymbolName());
         return rv.m_pVar;
       }
       rv = rv.m_pNext;
    }
    return null;
}

public void Run() throws Exception
{
    if (m_function.m_bBuiltIn)
    {
        if (m_function.m_sFunctionName.compareTo("coeff") == 0)
        {
            TermVariable t = (TermVariable)findRuntimeVariable("t");
            m_returnValue = t.m_fCoeff.evaluate().getCopy();
        } else if (m_function.m_sFunctionName.compareTo("degree") == 0)
        {
            if (m_function.m_aArguments.size() == 1)
            {
                m_returnValue = null;

                TermVariable t = (TermVariable)findRuntimeVariable("t");
                int i;
                LetterDegree ld;
                PMLVariable l,r;
                for (i = 0; i < t.m_aLetterDegrees.size(); i++)
                {
                    ld = (LetterDegree)t.m_aLetterDegrees.get(i);
                    if (m_returnValue == null)
                    {
                        r = ld.m_fDegree.evaluate().getCopy();
                        m_returnValue = r;
                    } else
                    {
                        l = m_returnValue.getCopy();
                        r = ld.m_fDegree.evaluate().getCopy();
                        m_returnValue = PMLMath.PlusOp_General(l, r, '+');
                    }
                }

                if (m_returnValue == null)
                {
                    m_returnValue = new IntVariable("tmp");
                    m_returnValue.setValue("0");
                }
            } else if (m_function.m_aArguments.size() == 2)
            {
                TermVariable t = (TermVariable)findRuntimeVariable("t");
                CharVariable c = (CharVariable)findRuntimeVariable("c");
                int i;
                LetterDegree ld;
                for (i = 0; i < t.m_aLetterDegrees.size(); i++)
                {
                    ld = (LetterDegree)t.m_aLetterDegrees.get(i);
                    if (ld.m_cLetter == c.m_cValue)
                    {
                        m_returnValue = ld.m_fDegree.evaluate().getCopy();
                        return;
                    }
                }

                m_returnValue = new IntVariable("tmp");
                m_returnValue.setValue("0");
```

```
                } else
                    throw new Exception("Invalid built-in function!\n" +
m_function.getPrototype());
            } else if (m_function.m_sFunctionName.compareTo("polyterm") == 0)
            {
                PolyVariable p = (PolyVariable)findRuntimeVariable("p");
                m_returnValue = new TermArray("tmp");
                int i;
                TermVariable t;
                for (i = 0; i < p.m_aTerms.size(); i++)
                {
                    t = (TermVariable)p.m_aTerms.get(i);
                    ((TermArray)m_returnValue).m_aTerms.add(t.getCopy());
                }
            } else if (m_function.m_sFunctionName.compareTo("length") == 0)
            {
                TermArray ta = (TermArray)findRuntimeVariable("ta");
                if (ta == null)
                {
                    CharArray ca = (CharArray)findRuntimeVariable("ca");
                    m_returnValue = new IntVariable("tmp");
                    m_returnValue.setValue(Integer.toString(ca.m_aChars.size()));
                } else
                {
                    m_returnValue = new IntVariable("tmp");
                    m_returnValue.setValue(Integer.toString(ta.m_aTerms.size()));
                }
            } else if ( m_function.m_sFunctionName.compareTo("variable") == 0)
            {
               PolyVariable plyA = (PolyVariable)findRuntimeVariable("pa");
               int i =0;
               TermVariable trmA;
               LetterDegree ldA;
               CharArray ca = new CharArray("ca");
               m_returnValue = new CharArray("car");
               for ( i = 0; i < plyA.m_aTerms.size(); i++)
               {
                   trmA = (TermVariable)plyA.m_aTerms.get(i);
                   int j = 0;
                   for ( j = 0; j < trmA.m_aLetterDegrees.size(); j++)
                   {
                        ldA =  (LetterDegree)trmA.m_aLetterDegrees.get(j);
                        CharVariable  c = new CharVariable("cv");
                        c.m_cValue =ldA.m_cLetter;
                        ca.m_aChars.add(c);
                   }
               }
               m_returnValue.setValue(ca);
            } else
                throw new Exception("Invalid built-in function!\n" +
m_function.getPrototype());
        } else
        {
            m_function.m_statementBlock.RunStatement();
        }
    }

    public PMLFunction m_function;
    public PMLVariable m_returnValue;
/**
    public  CharArray variable( PolyVariable plyA ) throws Exception
    {
        int i =0;
       TermVariable trmA = new TermVariable("trmA");
       LetterDegree ldA;
       //CharArray ca = new CharArray("ca");
       m_returnValue = new CharArray("ca");
       for ( i = 0; i < plyA.m_aTerms.size(); i++)
       {
           trmA = (TermVariable)plyA.m_aTerms.get(i);
           ldA =  (LetterDegree)trmA.m_aLetterDegrees.get(i);
           CharVariable  c = new CharVariable("cv");
           c.m_cValue =ldA.m_cLetter;
           ((CharArray)m_returnValue).m_aChars.add(c);
        }
```

```
   }
**/
}
```

PMLCondition.java

```
/*
 * PMLCondition.java
 *
 * Created on October 31, 2003, 7:39 PM
 */

/**
 *
 * @author  Shezan
 */
public class PMLCondition {

    static final int opEqualTo = 0;
    static final int opNotEqualTo = 1;
    static final int opLess = 2;
    static final int opLessEqual = 3;
    static final int opGreater = 4;
    static final int opGreaterEqual = 5;

    private PMLExpression m_pExprLeft;
    private PMLExpression m_pExprRight;
    private int m_nOperator;

    /** Creates a new instance of PMLCondition */
    public PMLCondition(PMLExpression r_pExprLeft, int r_nOperator, PMLExpression
r_pExprRight) {
        m_pExprLeft = r_pExprLeft;
        m_nOperator = r_nOperator;
        m_pExprRight = r_pExprRight;
    }

    public boolean evaluate() throws Exception {
        if (m_nOperator == PMLCondition.opEqualTo)
        {
            PMLVariable varLeft = m_pExprLeft.evaluate().getCopy();
            PMLVariable varRight = m_pExprRight.evaluate().getCopy();
            return varLeft.compare(varRight);
        } else if (m_nOperator == PMLCondition.opNotEqualTo)
        {
            PMLVariable varLeft = m_pExprLeft.evaluate().getCopy();
            PMLVariable varRight = m_pExprRight.evaluate().getCopy();
            return !varLeft.compare(varRight);
        }

        PMLVariable varLeft = PMLVariable.createVariable(PMLVariable.typeFloat, "temp");
        PMLVariable varRight = PMLVariable.createVariable(PMLVariable.typeFloat, "temp");
        varLeft.setValue(m_pExprLeft.evaluate());
        varRight.setValue(m_pExprRight.evaluate());
        float floatLeft = Float.parseFloat(varLeft.getStringValue());
        float floatRight = Float.parseFloat(varRight.getStringValue());
        switch (m_nOperator)
        {
            case PMLCondition.opLess:
                return (floatLeft < floatRight);
            case PMLCondition.opLessEqual:
                return (floatLeft <= floatRight);
            case PMLCondition.opGreater:
                return (floatLeft > floatRight);
            case PMLCondition.opGreaterEqual:
                return (floatLeft >= floatRight);
        }
        throw new Exception("Invalid condition operator");
    }
}
```

PMLStatement.java

```
/*
 * PMLStatement.java
 *
 * Created on October 21, 2003, 4:10 PM
 */

/**
 *
 * @author  Shezan
 */
public interface PMLStatement {
    public abstract void RunStatement() throws Exception;
}
```

PMLStatementAssign.java

```
/*
 * PMLStatementAssign.java
 *
 * Created on October 28, 2003, 1:54 PM
 */

/**
 *
 * @author  Shezan
 */
public class PMLStatementAssign implements PMLStatement {

    private PML_LValue m_pLValue;
    private PMLExpression m_pExpr;

    /** Creates a new instance of PMLStatementAssign */
    public PMLStatementAssign(PML_LValue r_pLValue, PMLExpression expr) {
        m_pLValue = r_pLValue;
        m_pExpr = expr;
    }

    public void RunStatement() throws Exception {
        PMLVariable var = m_pLValue.getLValue();
        var.setValue(m_pExpr.evaluate().getCopy());
    }

}
```

PMLStatementBlock.java

```
/*
 * PMLStatementBlock.java
 *
 * Created on October 27, 2003, 4:44 PM
 */

import java.util.*;

/**
 *
 * @author  Shezan
 */
public class PMLStatementBlock implements PMLStatement, PMLSymbolTable {

    /** Creates a new instance of PMLStatementBlock */
    public PMLStatementBlock() {
        m_aStatements = new Vector();
        m_aVariables = new Vector();
    }

    public void RunStatement() throws Exception {
        PMLFunctionInstance ar =
PMLRuntimeEnvironment.getRE().getCurrentActivationRecord();

        int i;
```

```
        for (i = 0; i < m_aVariables.size(); i++)
            ar.addRuntimeVariable(((PMLVariable)m_aVariables.get(i)).getCopy());

        for (i = 0; i < m_aStatements.size(); i++)
            ((PMLStatement)m_aStatements.get(i)).RunStatement();

        ar.removeRuntimeVariable(m_aVariables.size());
    }

    public void addVariable(PMLVariable var) throws Exception {
        if (findVariable(var.getSymbolName()) == null)
        {
            m_aVariables.add(var);
        } else
        {
            throw new Exception("Error: variable '" + var.getSymbolName() + "' already
defined!");
        }
    }

    public PMLVariable findVariable(String r_sSymbolName) {
        for (int i = 0; i < m_aVariables.size(); i++)
        {
            if
(((PMLVariable)m_aVariables.get(i)).getSymbolName().compareTo(r_sSymbolName) == 0)
                return (PMLVariable)m_aVariables.get(i);
        }

        return m_pSymbolTableParent.findVariable(r_sSymbolName);
    }

    PMLSymbolTable m_pSymbolTableParent;
    public void setParent(PMLSymbolTable parent) {
        m_pSymbolTableParent = parent;
    }

    public void addStatement(PMLStatement stmt)
    {
        m_aStatements.add(stmt);
    }

    private Vector m_aVariables;
    private Vector m_aStatements;

    public Vector getStatements()
    {
        return m_aStatements;
    }
}
```

PMLStatementBreak.java

```
/*
 * PMLStatementBreak.java
 *
 * Created on November 22, 2003, 12:43 AM
 */

/**
 *
 * @author  Shezan
 */
public class PMLStatementBreak implements PMLStatement {

    /** Creates a new instance of PMLStatementBreak */
    public PMLStatementBreak() {
    }

    public void RunStatement() throws Exception {
        throw new PMLBreakException();
    }

}
```

PMLStatementDo.java

```
/*
 * PMLStatementDo.java
 *
 * Created on October 31, 2003, 8:46 PM
 */

/**
 *
 * @author  Shezan
 */
public class PMLStatementDo implements PMLStatement {

    private PMLCondition m_pCond;
    private PMLStatement m_pStatement;
    /** Creates a new instance of PMLStatementDo */
    public PMLStatementDo(PMLCondition r_pCond, PMLStatement r_pStatement) {
        m_pCond = r_pCond;
        m_pStatement = r_pStatement;
    }

    public void RunStatement() throws Exception {
        do
        {
            try
            {
                m_pStatement.RunStatement();
            } catch (PMLBreakException e)
            {
                break;
            }
        } while (m_pCond.evaluate());
    }

}
```

PMLStatementExpr.java

```
/*
 * PMLStatementExpr.java
 *
 * Created on October 29, 2003, 11:06 AM
 */

/**
 *
 * @author  Shezan
 */
public class PMLStatementExpr implements PMLStatement {

    PMLExpression m_pExpr;

    /** Creates a new instance of PMLStatementExpr */
    public PMLStatementExpr(PMLExpression r_pExpr) {
        m_pExpr = r_pExpr;
    }

    public void RunStatement() throws Exception {
        m_pExpr.evaluate();
    }

}
```

PMLStatementIf.java

```
/*
 * PMLStatementIf.java
 *
 * Created on October 31, 2003, 7:56 PM
```

```
 */

/**
 *
 * @author  Shezan
 */
public class PMLStatementIf implements PMLStatement {

    private PMLCondition m_pCond;
    private PMLStatement m_pThen;
    private PMLStatement m_pElse;

    public PMLStatement getThenPart() { return m_pThen; }
    public PMLStatement getElsePart() { return m_pElse; }

    /** Creates a new instance of PMLStatementIf */
    public PMLStatementIf(PMLCondition r_pCond, PMLStatement r_pThen, PMLStatement
r_pElse) {
        m_pCond = r_pCond;
        m_pThen = r_pThen;
        m_pElse = r_pElse;
    }

    public void RunStatement() throws Exception {
        if (m_pCond.evaluate())
        {
            m_pThen.RunStatement();
        } else if (m_pElse != null)
        {
            m_pElse.RunStatement();
        }
    }

}
```

PMLStatementPrint.java

```
/*
 * PMLStatementPrint.java
 *
 * Created on October 31, 2003, 4:37 PM
 */

import java.util.*;

/**
 *
 * @author  Shezan
 */
public class PMLStatementPrint implements PMLStatement {

    private Vector m_aExprs; // array of expressions or string literals

    /** Creates a new instance of PMLStatementPrint */
    public PMLStatementPrint(Vector r_aExprs) {
        m_aExprs = r_aExprs;
    }

    public void RunStatement() throws Exception {
        int i;
        for (i = 0; i < m_aExprs.size(); i++)
        {
            if (m_aExprs.get(i).getClass() == String.class)
            {
                System.out.print((String)m_aExprs.get(i));
            } else
            {

System.out.print(((PMLExpression)m_aExprs.get(i)).evaluate().getStringValue());
            }
        }
        System.out.println("");
    }
```

```
}
```

PMLStatementReturn.java

```
/*
 * PMLStatementReturn.java
 *
 * Created on October 31, 2003, 4:56 PM
 */

/**
 *
 * @author  Shezan
 */
public class PMLStatementReturn implements PMLStatement {

    private PMLExpression m_pExpr;
    /** Creates a new instance of PMLStatementReturn */
    public PMLStatementReturn(PMLExpression r_pExpr) {
        m_pExpr = r_pExpr;
    }

    public void RunStatement() throws Exception {
        if (m_pExpr != null)

PMLRuntimeEnvironment.getRE().getCurrentActivationRecord().m_returnValue.setValue(m_pExpr.
evaluate());
        throw new PMLReturnException();
    }

}
```

PMLStatementWhile.java

```
/*
 * PMLStatementWhile.java
 *
 * Created on October 31, 2003, 8:34 PM
 */

/**
 *
 * @author  Shezan
 */
public class PMLStatementWhile implements PMLStatement {

    private PMLCondition m_pCond;
    private PMLStatement m_pStatement;
    /** Creates a new instance of PMLStatementWhile */
    public PMLStatementWhile(PMLCondition r_pCond, PMLStatement r_pStatement) {
        m_pCond = r_pCond;
        m_pStatement = r_pStatement;
    }

    public void RunStatement() throws Exception {
        while (m_pCond.evaluate())
        {
            try
            {
                m_pStatement.RunStatement();
            } catch (PMLBreakException e)
            {
                break;
            }
        }
    }

}
```

PMLExpression.java

```
/*
 * PMLExpression.java
 *
 * Created on October 21, 2003, 4:18 PM
 */

/**
 *
 * @author  Shezan
 */
public interface PMLExpression {
    public PMLVariable evaluate() throws Exception;
}
```

PMLExprArith.java

```
/*
 * PMLExprArith.java
 *
 * Created on November 11, 2003, 2:04 PM
 * modified : November 22, 2003 --  Hari Kurup
 * PlusOp_General called with 3 parameters, including operator
 */

/**
 *
 * @author  Shezan
 */
public class PMLExprArith implements PMLExpression {

    PMLExpression m_pExprLeft;
    PMLExpression m_pExprRight;
    char m_cOperator;

    /** Creates a new instance of PMLExprArith */
    public PMLExprArith(PMLExpression exprLeft, char op, PMLExpression exprRight) {
        m_pExprLeft = exprLeft;
        m_pExprRight = exprRight;
        m_cOperator = op;
    }

    public PMLVariable evaluate() throws Exception {
        PMLVariable left = m_pExprLeft.evaluate().getCopy();
        PMLVariable right = m_pExprRight.evaluate().getCopy();
//        char temp[] = { m_cOperator };
//        PMLDebug.shez_println("Operator " + new String(temp) + ": " +
left.getStringValue() + " and " + right.getStringValue(), 2);
        PMLVariable ret = PMLMath.PlusOp_General(left, right,m_cOperator);
//        PMLDebug.shez_println("--> I got " + ret.getStringValue() + " -> which is (" +
PMLVariable.types[ret.getVariableType()] + ")\n", 2);
        return ret;
    }

}
```

PMLExprFuncCall.java

```
/*
 * PMLExprFuncCall.java
 *
 * Created on October 29, 2003, 10:45 AM
 */

import java.util.*;

/**
 *
 * @author  Shezan
 */
public class PMLExprFuncCall implements PMLExpression {
```

```
    private String m_sFunctionName;
    private Vector m_aParamExprs;

    /** Creates a new instance of PMLExprFuncCall */
    public PMLExprFuncCall(String r_sFunctionName, Vector r_aParamExprs) {
        m_sFunctionName = r_sFunctionName;
        m_aParamExprs = r_aParamExprs;
    }

    public PMLVariable evaluate() throws Exception {

        // evaluate all param expressions and store the results
        // in t_aParams
        Vector t_aParams = new Vector();
        int i;
        for (i = 0; i < m_aParamExprs.size(); i++)
            t_aParams.add(((PMLExpression)m_aParamExprs.get(i)).evaluate());

        return PMLRuntimeEnvironment.getRE().executeFunction(m_sFunctionName, t_aParams);
    }

}
```

PMLExprLiteral.java

```
/*
 * PMLExprLiteral.java
 *
 * Created on October 28, 2003, 1:48 PM
 */

/**
 *
 * @author  Shezan
 */
public class PMLExprLiteral implements PMLExpression {

    private PMLVariable m_pVar;

    /** Creates a new instance of PMLExprLiteral */
    public PMLExprLiteral(PMLVariable var) {
        m_pVar = var;
    }

    public PMLVariable evaluate() throws Exception {
        return m_pVar;
    }

}
```

PMLExprTypecast.java

```
/*
 * PMLExprTypecast.java
 *
 * Created on November 29, 2003, 3:13 PM
 */

/**
 *
 * @author  Shezan
 */
public class PMLExprTypecast implements PMLExpression {

    private PMLVariable m_pVar;
    private PMLExpression m_pExpr;
    /** Creates a new instance of PMLExprTypecast */
    public PMLExprTypecast(PMLVariable r_pVar, PMLExpression r_pExpr) {
        m_pVar = r_pVar;
        m_pExpr = r_pExpr;
    }
```

```
    public PMLVariable evaluate() throws Exception {
        m_pVar.setValue(m_pExpr.evaluate().getCopy(), true);
        return m_pVar;
    }

}
```

PML_LValue.java

```
/*
 * PML_LValue.java
 *
 * Created on November 12, 2003, 4:11 PM
 */

/**
 *
 * @author  Shezan
 */
public interface PML_LValue {
    public PMLVariable getLValue() throws Exception;
}
```

PMLExprVariable.java

```
/*
 * PMLExprVariable.java
 *
 * Created on October 28, 2003, 10:36 AM
 */

/**
 *
 * @author  Shezan
 */
public class PMLExprVariable implements PMLExpression, PML_LValue {

    private String m_sSymbolName;
    /** Creates a new instance of PMLExprVariable */
    public PMLExprVariable(String r_sSymbolName) {
        m_sSymbolName = r_sSymbolName;
    }

    public String getSymbolName() { return m_sSymbolName; }

    public PMLVariable evaluate() throws Exception {
        PMLVariable var =
PMLRuntimeEnvironment.getRE().findRuntimeVariable(m_sSymbolName);
        if (var == null)
            throw new Exception("Unable to evaluate variable '" + m_sSymbolName + "'");
        return var;
    }

    public PMLVariable getLValue() throws Exception {
        return evaluate();
    }

}
```

PMLExprVariableArray.java

```
/*
 * PMLExprVariableArray.java
 *
 * Created on November 12, 2003, 2:20 PM
 */

/**
 *
 * @author  Shezan
 */
```

```
public class PMLExprVariableArray implements PMLExpression, PML_LValue {
    private PMLExprVariable m_var;
    private PMLExpression m_index;
    public PMLExprVariableArray(String r_sSymbolName, PMLExpression r_index)
    {
        m_var = new PMLExprVariable(r_sSymbolName);
        m_index = r_index;
    }

    public PMLVariable evaluate() throws Exception {
        PMLVariable var = m_var.evaluate();

        IntVariable i = new IntVariable("tmp");
        i.setValue(m_index.evaluate(), true);
        i.m_nValue--;

        try
        {
            if (var.getVariableType() == PMLVariable.typeTermArray)
                return (PMLVariable)((TermArray)var).getTerm(i.m_nValue);
            else if (var.getVariableType() == PMLVariable.typeCharArray)
                return (PMLVariable)((CharArray)var).getTerm(i.m_nValue);
        } catch(Exception e)
        {
            throw new Exception("Invalid array subscript");
        }

        return null;
    }

    public PMLVariable getLValue() throws Exception {
        return evaluate();
    }

}
```

PMLDebug.java

```
/*
 * PMLDebug.java
 *
 * Created on November 11, 2003, 10:06 AM
 */

/**
 *
 * @author  Shezan
 */
public class PMLDebug {

    public static void println(String s) {
//        System.out.println(s);
    }

    public static void shez_println(String s, int level) {
        if (true) return;
        if (//true ||
            level == 1 // parsing,walking,executing
            || level == 2 // Adding & I got
            //|| level == 3 // running func proto
                       )
            System.out.println("DEBUG: " + s);
    }

}
```

PMLBreakException.java

```
/*
 * PMLBreakException.java
 *
 * Created on November 22, 2003, 12:44 AM
```

```
 */

/**
 *
 * @author  Shezan
 */
public class PMLBreakException extends Exception {

    /** Creates a new instance of PMLBreakException */
    public PMLBreakException() {
    }

}
```

PMLReturnException.java

```
/*
 * PMLReturnException.java
 *
 * Created on October 31, 2003, 5:02 PM
 */

/**
 *
 * @author  Shezan
 */
public class PMLReturnException extends Exception {

    /** Creates a new instance of PMLReturnException */
    public PMLReturnException() {
    }

}
```

PMLNoValidConversionException.java

```
/*
 * PMLNoValidConversionException.java
 *
 * Created on October 19, 2003, 1:56 PM
 */

/**
 *
 * @author  Shezan
 */
public class PMLNoValidConversionException extends Exception {

    /** Creates a new instance of PMLNoValidConversionException */
    public PMLNoValidConversionException(PMLVariable to, PMLVariable from) {
        super("No valid conversion from "
                  + PMLVariable.types[from.getVariableType()]
                  + " to " + PMLVariable.types[to.getVariableType()]);
    }
}
```

PMLMath.java (backend)

```
/*****************************************************************************/
/***      PMLMath class - handles mathematical operations                  */
/***      for PML                                                          */
/***      07-Nov-2003  Hari Kurup : Released with addition                 */
/***                                features                               */
/***      09-Nov-2003  Hari Kurup : Termarray additions                    */
/***      11-Nov-2003 Shezan Baig : added PlusOp_General                   */
/***      15-Nov-2003  Hari Kurup : modified term addition to handle       */
/***                                PMLExpression coefficients. Also       */
/***                                removed some redundant PlusOp funcs    */
/***      22-Nov-2003  Hari Kurup : modified PlusOps to handle '-','*' &   */
/***                                '/'. The latter two operations are     */
/***                                valid for int and floats only.         */
```

```
/***      23-Nov-2003  Hari Kurup : added '-' operation to poly and term-     */
/***                                 arrays.                                  */
/***      27-Nov-2003  Hari Kurup : added exception for '*' & '/' operation   */
/***                                 for non-float/int datatypes. Modified    */
/***                                 subtraction logic for terms & polys.     */
/***                                 Terms in a poly with zero coeff are      */
/***                                 eliminated. Subtraction of like char     */
/***                                 coeffs result in 0.                      */
/***      29-Nov-2003  Hari Kurup : fixed bug in comparing terms with 0 deg   */
/***      01-Dec-2003  Hari Kurup : CharArray and Char additions              */
/***
/***      08-Dec-2003 Shezan : Fixed TermDegreeCompare() function
 *                            Fixed RemoveZeroCoeff() function
 *                            Fixed '-' operator for chararray & termarray
/*****************************************************************************/

import java.util.*;
class PMLMath
{
    int m_intVarType;
       int value;

        /**** SHEZAN ADDED 11 Nov'03 *****/
        public static PMLVariable PlusOp_General(PMLVariable a, PMLVariable b, char chrOp)
throws Exception
        {
            PMLVariable ret = null;
            char temp[] = { chrOp };
            PMLDebug.shez_println("Operator " + new String(temp) + ": " +
a.getStringValue() + " (" + PMLVariable.types[a.getVariableType()] + ") and " +
b.getStringValue() + " (" + PMLVariable.types[b.getVariableType()] + ")", 2);

            if (ret == null)
            {
                // try int,int
                try
                {
                    IntVariable inta = new IntVariable("where"), intb = new
IntVariable("is");
                    inta.setValue(a); intb.setValue(b);
                    ret = PlusOp(inta, intb,chrOp);
                    PMLDebug.shez_println("Used int,int", 2);
                } catch(Exception e) { }
            }
            if (ret == null)
            {
                // try float,float
                try
                {
                    FloatVariable floata = new FloatVariable("the"), floatb = new
FloatVariable("pizza?");
                    floata.setValue(a); floatb.setValue(b);
                    ret = PlusOp(floata, floatb,chrOp);
                    PMLDebug.shez_println("Used float,float", 2);
                } catch(Exception e) { }
            }
            if (ret == null)
            {
                if ( ( chrOp == '+' ) || ( chrOp == '-' ) )
                {
                    // try term,term
                    if (ret == null)
                    {
                        try
                        {
                            TermVariable terma = new TermVariable("bla"), termb = new
TermVariable("bla");
                            terma.setValue(a); termb.setValue(b);
                            ret = PlusOp(terma, termb,chrOp);
                            PMLDebug.shez_println("Used term,term", 2);
                        } catch(Exception e) {  }
                    }
                    if (ret == null)
                    {
                        // try poly,poly
```

```
                        try
                        {
                            PolyVariable polya = new PolyVariable("bla"), polyb = new
PolyVariable("bla");
                            polya.setValue(a); polyb.setValue(b);
                            ret = PlusOp(polya, polyb,chrOp);
                            PMLDebug.shez_println("Used poly,poly", 2);
                        } catch(Exception e) { }
                    }
                    if (ret == null)
                    {
                        try
                        {
                            TermArray ta = new TermArray("ta");
                            TermVariable trmA = new TermVariable("trm");
                            ta.setValue(a);
                            trmA.setValue(b);
                            ret = PlusOp(ta,trmA, chrOp);
                            PMLDebug.shez_println("Used termarray,term", 2);
                        } catch(Exception e){}
                    }
                    if (ret == null)
                    {
                        try
                        {
                            TermArray ta = new TermArray("ta"),  tb = new TermArray("tb");
                            ta.setValue(a); tb.setValue(b);
                            ret = PlusOp(ta, tb, chrOp);
                            PMLDebug.shez_println("Used termarray,termarray", 2);
                        } catch(Exception e){}
                    }
                    if (ret == null)
                    {
                        try
                        {
                            CharArray ca = new CharArray("ca");
                            CharVariable chrA = new CharVariable("chr");
                            ca.setValue(a);
                            chrA.setValue(b);
                            ret = PlusOp(ca,chrA, chrOp);
                            PMLDebug.shez_println("Used chararray,char", 2);
                        } catch (Exception e){}
                    }
                    if (ret == null)
                    {
                        try
                        {
                            CharArray ca = new CharArray("ca"), cb = new CharArray("cb");
                            ca.setValue(a); cb.setValue(b);
                            ret = PlusOp(ca,cb,chrOp);
                            PMLDebug.shez_println("Used chararray,chararray", 2);
                        } catch ( Exception e) {}
                    }
                } else
                {
                    if ( (chrOp == '*') || (chrOp == '/' ))
                        throw new Exception("operation " + chrOp + " not defined for non-
int/float datatypes" );
                }
            }

            if (ret != null)
            {
                PMLDebug.shez_println("--> I got " + ret.getStringValue() + " -> which is
(" + PMLVariable.types[ret.getVariableType()] + ")\n", 2);
                return ret;
            } else
                // nothing worked... throw an exception
                throw new Exception("Operator " + new String(temp) + " undefined for " +
a.getStringValue() + " (" + PMLVariable.types[a.getVariableType()] + ") and " +
b.getStringValue() + " (" + PMLVariable.types[b.getVariableType()] + ")");
        }
        /**** END - SHEZAN's PART 11 Nov '03 ****/
```

```
// addition of two integers -- HK
       public static IntVariable PlusOp ( IntVariable a, IntVariable b, char chrOp )
throws Exception
       {
           int c = 0;
            String s;
            IntVariable intVarResult = new IntVariable( "result");
            switch(chrOp)
            {
                case '+' :
                          c = Integer.parseInt(a.getStringValue()) +
Integer.parseInt(b.getStringValue());
                          break;
                case '-' :
                          c = Integer.parseInt(a.getStringValue()) -
Integer.parseInt(b.getStringValue());
                          break;
                case '*' :
                          c = Integer.parseInt(a.getStringValue()) *
Integer.parseInt(b.getStringValue());
                          break;
                case '/' :
                          c = Integer.parseInt(a.getStringValue()) /
Integer.parseInt(b.getStringValue());
                          break;
            }
            s = "" + c;
            intVarResult.setValue( s );
            PMLDebug.println("here i am ");
           return intVarResult;
    }


// addition of two floats -- HK
       public static FloatVariable PlusOp ( FloatVariable a, FloatVariable b, char chrOp
) throws Exception
       {
           float c = 0;
            String s;
            FloatVariable fltVarResult = new FloatVariable( "result");
            switch(chrOp)
            {
                case '+' :
                          c = Float.parseFloat(a.getStringValue()) +
Float.parseFloat(b.getStringValue());
                         break;
              case '-' :
                          c = Float.parseFloat(a.getStringValue()) -
Float.parseFloat(b.getStringValue());
                        break;
              case '*' :
                          c = Float.parseFloat(a.getStringValue()) *
Float.parseFloat(b.getStringValue());
                        break;
              case '/' :
                          c = Float.parseFloat(a.getStringValue()) /
Float.parseFloat(b.getStringValue());
                        break;
            }
            s = "" + c;
            fltVarResult.setValue( s );
           return fltVarResult;
    }


// addition of two terms -- HK
        public static PMLVariable PlusOp ( TermVariable a, TermVariable b, char chrOp )
throws Exception
       {
           float fltCoeffA , fltCoeffB;
           //char chrCoeffA, chrCoeffB;
           String strCoeffA;
           String strCoeffB;
           float fltDegA =0.0f;
```

```
            float fltDegB =0.0f;
             String s;
            char chrVarA, chrVarB;
             int i = 0, j = 0;
            boolean intFound = false;
            LetterDegree ldA, ldB;
             FloatVariable fvCoeffA = new FloatVariable("tmp1");
             FloatVariable fvCoeffB = new FloatVariable("tmp2");
            CharVariable cvCoeffA = new CharVariable("tmp1");
            CharVariable cvCoeffB = new CharVariable("tmp2");
             PMLVariable pmlA =  a.m_fCoeff.evaluate() ;
            PMLVariable pmlB = b.m_fCoeff.evaluate();
            PMLDebug.println("term a =" + pmlA.getStringValue());
            PMLDebug.println("term b =" + pmlB.getStringValue());


/****************
  ************/
            PolyVariable plyVarResult = new PolyVariable("polyresult");

             TermVariable trmVarResult = new TermVariable( "result");
             TermVariable trmTempA = new TermVariable( "tempA");
             TermVariable trmTempB = new TermVariable( "tempB");

             PMLDebug.println("term result = " + a.getStringValue() );
             PMLDebug.println("term result = " + b.getStringValue() );

             //LetterDegree ldA = new LetterDegree();
             //LetterDegree ldB = new LetterDegree();
             FloatVariable fvDegreeA = new FloatVariable("d1");
             FloatVariable fvDegreeB = new FloatVariable("d2");


             if (  a.m_aLetterDegrees.size() == b.m_aLetterDegrees.size() )
             {
                if ( TermDegreeCompare(a,b))
                intFound = true;
                else
                intFound = false;
                /**
                 ldA = (LetterDegree)a.m_aLetterDegrees.get(i);
                 ldB = (LetterDegree)b.m_aLetterDegrees.get(i);
                if ( ldA.compare(ldB) )
                **/
             }
            else
                intFound = false;

             if ( intFound )
             {
                for( i = 0; i <a.m_aLetterDegrees.size(); i++)

trmVarResult.m_aLetterDegrees.add((LetterDegree)a.m_aLetterDegrees.get(i));
                switch (chrOp)
                {
                case '+' :
                          trmVarResult.m_fCoeff = new
PMLExprLiteral(PlusOp_General(pmlA,pmlB, chrOp));
                        break;
                case '-' :
                        PMLVariable pmlC = pmlB.negate();
                        if ( a.compare(b) )
                        {
                            IntVariable intVarA = CreateZeroTerm();
                            IntVariable intVarB = CreateZeroTerm();
                            trmVarResult.m_fCoeff = new
PMLExprLiteral(PlusOp_General(intVarA, intVarB,'+'));
                        }
                        else
                          trmVarResult.m_fCoeff = new
PMLExprLiteral(PlusOp_General(pmlA,pmlC, '+'));
                        break;
                 }

                 PMLDebug.println("added==");
```

```
                return trmVarResult;

            }
            else
            {

                plyVarResult.m_aTerms.add(a);
                 if ( chrOp == '-')
                plyVarResult.m_aTerms.add(b.negate());
                 else
                plyVarResult.m_aTerms.add(b);

                 return RemoveZeroCoeff(plyVarResult);
            }
   }

// addition of 2 polys  -- HK
        public static PolyVariable PlusOp ( PolyVariable a, PolyVariable b, char chrOp)
throws Exception
        {
            // added by shezan - begin (8 Dec)
            if (a.m_aTerms.size() == 0)
                return RemoveZeroCoeff(b);
            else if (b.m_aTerms.size() == 0)
                return RemoveZeroCoeff(a);
            // shezan - end

            int i;
            TermVariable trmTemp;
            TermVariable trmLast;
            PolyVariable plyVarRes = new PolyVariable("res");
            PolyVariable plyVarTemp = new PolyVariable("res");

            PMLDebug.println("---poly A---" + b.m_aTerms.size() );
            for( i = 0; i < a.m_aTerms.size(); i++)
            {
                trmTemp = (TermVariable)a.m_aTerms.get(i);
                PMLDebug.println("---term A ---" + trmTemp.getStringValue() );
                plyVarRes.m_aTerms.add( trmTemp);
            }
            trmLast = (TermVariable)a.m_aTerms.get(a.m_aTerms.size()-1);
            if ( chrOp == '-' )
            {
                for( i = 0; i < b.m_aTerms.size(); i++)
                {
                    trmTemp = (TermVariable)b.m_aTerms.get(i);

                    if ( ( i== 0) && (TermDegreeCompare(trmLast, trmTemp ) ))
                    {
                         //PMLVariable pmlB = trmTemp.negate();
                       //trmTemp.setValue(pmlB);
                         PMLVariable pmlA = PlusOp_General(trmLast, trmTemp, chrOp);
                         trmTemp.setValue(pmlA);
                         plyVarRes.m_aTerms.remove(a.m_aTerms.size()-1);

                    }

                   if (( i > 0 ) || ( !TermDegreeCompare(trmLast,trmTemp) ))
                   {
                        PMLVariable pmlB = trmTemp.negate();
                      trmTemp.setValue(pmlB);
                    }
                        PMLDebug.println("---term B ---" + trmTemp.getStringValue() );
                        plyVarRes.m_aTerms.add(trmTemp );
                        PMLDebug.println(" the poly value = " + plyVarRes.getStringValue()
);
                }
            }
            else
            {
                for( i = 0; i < b.m_aTerms.size(); i++)
                {
                    trmTemp = (TermVariable)b.m_aTerms.get(i);
                    if ( ( i== 0) && (TermDegreeCompare(trmLast, trmTemp ) ))
                    {
```

```
                        PMLVariable pmlA = PlusOp_General(trmLast, trmTemp, chrOp);
                        trmTemp.setValue(pmlA);
                        plyVarRes.m_aTerms.remove(a.m_aTerms.size()-1);
                    }

                    PMLDebug.println("---term B ---" + trmTemp.getStringValue() );
                    plyVarRes.m_aTerms.add( trmTemp);
                    PMLDebug.println(" the poly value = " + plyVarRes.getStringValue() );
                }
            }
            return RemoveZeroCoeff(plyVarRes);
        }

        public static IntVariable CreateZeroTerm( ) throws Exception
        {
            IntVariable intVarA = new IntVariable("a");
//            String s = "";
//            s = s+0;
//            intVarA.setValue(s);
            intVarA.m_nValue = 0;
            return intVarA;
        }
        private static PolyVariable RemoveZeroCoeff( PolyVariable pmlA) throws Exception
        {
            /*
            TermVariable trmTemp = new TermVariable("t");
           TermVariable trmPoly = new TermVariable("tp");
            IntVariable intVarA = CreateZeroTerm();
            IntVariable intVarB = CreateZeroTerm();
           PolyVariable plyResult = new PolyVariable("result");
           int i = 0;
            trmTemp.m_fCoeff = new PMLExprLiteral(PlusOp_General( intVarA, intVarB,'+'));
           for( i =0; i < pmlA.m_aTerms.size(); i++)
           {
               trmPoly = (TermVariable)pmlA.m_aTerms.get(i);
               if ( trmPoly.compare(trmTemp) )
               {
                   continue;
                }
               else
               {
                   plyResult.m_aTerms.add(trmPoly);
                }
            }
             */

            // modified by Shezan (8 Dec)
            IntVariable zero = CreateZeroTerm();
            PolyVariable plyResult = new PolyVariable("tmp");
            TermVariable t;

            int i;
            for (i = 0; i < pmlA.m_aTerms.size(); i++)
            {
                t = (TermVariable)pmlA.m_aTerms.get(i);
                if (!t.m_fCoeff.evaluate().compare(zero))
                    plyResult.m_aTerms.add(t);
            }

            return plyResult;
        }

// method to compare degrees -- HK
        private static boolean TermDegreeCompare (TermVariable trmA, TermVariable trmB )
throws Exception
        {
            // modified by Shezan 8 Dec
            if (trmA.m_aLetterDegrees.size() != trmB.m_aLetterDegrees.size())
                return false;

             int i,j;
             LetterDegree ld1, ld2;
             boolean checked[] = new boolean[trmA.m_aLetterDegrees.size()];
             for (i = 0; i < trmA.m_aLetterDegrees.size(); i++)
                 checked[i] = false;
```

```
            for (i = 0; i < trmA.m_aLetterDegrees.size(); i++)
            {
                ld1 = (LetterDegree)trmA.m_aLetterDegrees.get(i);
                for (j = 0; j < trmB.m_aLetterDegrees.size(); j++)
                {
                    if (checked[j])
                        continue;

                    ld2 = (LetterDegree)trmB.m_aLetterDegrees.get(j);
                    if (ld1.compare(ld2))
                    {
                        checked[j] = true;
                        break;
                    }
                }
                if (j == trmB.m_aLetterDegrees.size())
                    return false;
            }

            return true;



           /*
           int i =0;
          int j = 0;
          LetterDegree ldA, ldB;
          boolean bolFound = false;

           if ( (trmA.m_aLetterDegrees.size() == 0 ) && (trmB.m_aLetterDegrees.size()==0))
           bolFound = true;

          for ( i =0 ; i < trmA.m_aLetterDegrees.size(); i++)
          {
             ldA = (LetterDegree)trmA.m_aLetterDegrees.get(i);
             for( j = 0; j < trmB.m_aLetterDegrees.size(); j++)
             {
                 ldB = (LetterDegree)trmB.m_aLetterDegrees.get(j);
                 if ( ldB.compare(ldA) )
                {
                    bolFound = true;
                    break;
                }
             }
             if (! bolFound )
             break;
           }
          return bolFound;
            */
       }


// addition of 2 termarrays -- HK
        public static TermArray PlusOp( TermArray a, TermArray b,char chrOp) throws
Exception
       {
           TermArray  ta = new TermArray("temp");
           TermVariable tempA = new TermVariable("tempA");
           TermVariable tempB = new TermVariable("tempB");
           boolean blFound;
           int intIndex = 0;
           int i = 0;
           int j = 0;
           if ( chrOp == '+' )
           {
           for( i = 0; i < a.m_aTerms.size(); i++)
           {
               ta.m_aTerms.add((TermVariable)a.m_aTerms.get(i));
            }
           for( i =0; i <b.m_aTerms.size(); i++)
           {
               ta.m_aTerms.add((TermVariable)b.m_aTerms.get(i));
            }
           }
```

```
            else
            {
                int[] intIndexMap = new int[b.m_aTerms.size()];

                for( i =0; i <b.m_aTerms.size(); i++)
                {
                    intIndexMap[i] = 0;
                 }

                for( i = 0; i < a.m_aTerms.size(); i++)
                {
                   tempA = (TermVariable)a.m_aTerms.get(i);
                   blFound = false;
                    for( j =0; j <b.m_aTerms.size(); j++)
                    {
                       tempB = (TermVariable)b.m_aTerms.get(j);
                       if ( intIndexMap[j] == 1 )
                       continue;
                       if (! tempB.compare(tempA) )
                       {
                          continue;
                        }
                       else
                       {
                          blFound = true;
                          intIndex = j;
                          intIndexMap[j] = 1;
                          break;
                        }
                     }
                     if( !blFound )
                     ta.m_aTerms.add((TermVariable)a.m_aTerms.get(i));
                 }
            }
            return ta;
        }

// addition of termarray and a term -- HK
        public static TermArray PlusOp( TermArray a, TermVariable b, char chrOp ) throws
Exception
        {
            TermArray ta = new TermArray("temp");
            int i =0;
            if ( chrOp == '+' )
            {
                 /*
                 trmTemp.m_fCoeff = b.m_fCoeff;
                 for( i =0; i < b.m_aLetterDegrees.size(); i++)
                     trmTemp.m_aLetterDegrees.add((LetterDegree)b.m_aLetterDegrees.get(i));
                  */
                 for(i=0; i< a.m_aTerms.size(); i++)
                     ta.m_aTerms.add((TermVariable)a.m_aTerms.get(i));
                 ta.m_aTerms.add(b);
            }
            else
            {
                TermVariable trmTemp;
                 boolean removed = false;
                 for(i=0; i< a.m_aTerms.size(); i++)
                {
                    trmTemp = (TermVariable)a.m_aTerms.get(i);
                    if (!removed && b.compare(trmTemp))
                     {
                         removed = true;
                     } else
                    {
                         ta.m_aTerms.add(trmTemp);
                     }
                 }
              }
            return ta;
        }
```

```
        public static CharArray PlusOp( CharArray a, CharArray b, char chrOp) throws
Exception
        {
            CharArray ca   = new CharArray("res");
           int i =0;

           if ( chrOp == '+' )
           {
                for( i = 0; i < a.m_aChars.size(); i++)
                {
                    ca.m_aChars.add((CharVariable)a.m_aChars.get(i));
                }
                for( i =0; i <b.m_aChars.size(); i++)
                {
                    ca.m_aChars.add((CharVariable)b.m_aChars.get(i));
                }
           } else if (chrOp == '-') // added by shezan (8 Dec)
            {
               int[] intIndexMap = new int[b.m_aChars.size()];

                CharVariable tempA, tempB;
               for( i =0; i <b.m_aChars.size(); i++)
               {
                   intIndexMap[i] = 0;
                }

                boolean blFound;
                int j;
               for( i = 0; i < a.m_aChars.size(); i++)
               {
                  tempA = (CharVariable)a.m_aChars.get(i);
                  blFound = false;
                   for( j =0; j <b.m_aChars.size(); j++)
                   {
                      tempB = (CharVariable)b.m_aChars.get(j);
                      if ( intIndexMap[j] == 1 )
                      continue;
                      if (! tempB.compare(tempA) )
                      {
                         continue;
                       }
                      else
                      {
                         blFound = true;
                         intIndexMap[j] = 1;
                         break;
                       }
                    }
                    if( !blFound )
                        ca.m_aChars.add(tempA);
                }
            }
           return ca;
        }
     public static CharArray PlusOp( CharArray a, CharVariable b, char chrOp) throws
Exception
        {
            CharArray ca   = new CharArray("res");
           int i =0;

           if ( chrOp == '+' )
           {
                for( i = 0; i < a.m_aChars.size(); i++)
                    ca.m_aChars.add((CharVariable)a.m_aChars.get(i));
               ca.m_aChars.add(b);
           } else if (chrOp == '-') // added by Shezan (8 Dec)
            {
                boolean removed = false;
                for (i = 0; i < a.m_aChars.size(); i++)
                    if (!removed && ((CharVariable)a.m_aChars.get(i)).compare(b))
                        removed = true;
                    else
                        ca.m_aChars.add(((CharVariable)a.m_aChars.get(i)));
            }
```

```
            return ca;
        }

    }
```

Bibliography:

[1].Software unit testing. Rodney Parkin, IV&V Australia