Language Reference Manual for: Polynomial Manipulation Language (PML)

Authors: Melinda Agyekum (mya2001@columbia.edu)
Shezan Baig (sb2284@columbia.edu)
Hari Kurup (hgk2101@columbia.edu) – Group Leader
Subadhra Sridharan (ss2355@columbia.edu)

INTRODUCTION

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.

2. 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.

2.1 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 }#
```

2.2 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

2.3 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	
else	int	termarray	

2.4 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.

2.4.1 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

2.4.2 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,

2.4.3 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)*

2.4.4 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

2.4.5 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

2.4.6 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.

2.4.7 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'.

2.4.8 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. CONVERSIONS

Implicit type conversions will be supported for the following:

int → float
float → int (fraction portion rounded and discarded)
float → term
term → poly
char → term

The following explicit conversions are available:

1. (poly -> termarray) which is done explicitly using the polyterm() function
2. term -> float is converted using the coeff(…) method.

4. EXPRESSIONS

4.1 Identifiers

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

4.2 *(expression)*

A parenthesized expression is identical to an expression without parenthesis.

4.3 Operators

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

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

4.3.1.1 *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

4.3.1.2 *expr / expr* is an expression implying division. Multiplication conversion rules from section 4.3.1.2 apply.

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

4.3.2 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.

4.3.2.1 *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.

4.3.2.2 *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`.

4.3.3 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)

4.3.3.1 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)

4.34 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

5. 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

6. STATEMENTS

Most statements are expression statements of the form:

Expression;

6.1 Compound Statement

Several statements can be used in place of one statement.

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

statement-list:
        statement
        statement statement-list
```

6.2 Conditional Statement

Two forms of conditional statement are:

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

6.3 Loop statements

Two forms of loop statements are while and do while.

While (expression) statement end

do statement while (expression) end

6.4 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;
```

6.5 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.

6.6 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.

7. 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;
```

8. SCOPE RULES

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

8.1 – 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.

8.2 – 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.

8.3 – 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().

9. 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.

10. 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.

11. 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.

11.1 – 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
```

11.2 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.

APPENDIX A

A1. SAMPLE CODE TO ADD TWO POLYNOMIALS

```
func void main
begin
      vars
              poly p1 = 2X^2 + 4X;
              poly p2 = 4X^2 – 2X ;
              poly temp;
              poly sum;
              int i;
              termarray t1;
              termarray t2;
     end

     t1 = polyterm(p1);
     t2 = polyterm(p1);

     i = length(t1);
     while( i >= 1 )
     begin

             # add the two terms
             temp = t1[i] + t2[i];

             # concatenate the polynomials to form the complete result
             sum = sum + temp;
             i = i – 1;
     end
     print sum ;
end
```

The *polyterm(..)* operator returns an array of individual terms from the polynomial . So after `t1 = polyterm(p1);` t1 will have two elements which are *2X^2* and *4X*. `t2` will have *4X^2* and *-2X*. The elements in these arrays can be accessed by the array notation *array[index]* . The index count starts from 1.

Length...() is an operator that takes an array and returns the count of the number of elements in an array.

APPENDIX B

B1. 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 lcoeff(…) : This function takes a polynomial and returns the leading coefficient of the polynomial. The leading coefficient is the coefficient of the term within the polynomial with the highest degree. Valid invocation of the function is as follows:

lcoeff(poly)

example: `poly p = 2x^3 + 3x + 4; lcoeff( p );` ***Return Value*** 2
`poly p =  ax^2 + bx + c; lcoeff( p );` ***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