# Polynomial Language Project

Darrell Bethea

Michael Dougherty

Santosh Thammana

Mohit Vazirani

# Language Reference Manual

## 1 Lexical conventions

### 1.1 Comments

The characters "//" denote a single-line comment.

### 1.2 Identifiers

An identifier consists of letters, digits and underscores "_". However, the first character of an identifier must be a either a letter or an underscore. Identifiers are case – sensitive.

### 1.3 Keywords

The following identifiers are reserved as keywords:

| | | | | | |
|---|---|---|---|---|---|
| **if** | **else** | **and** | **function** | | **not** |
| **return** | **exit** | **while** | **prototype** | **or** | |

### 1.4 Numbers

A number can be either an integer or a float. An integer consists of one or more digits. A float consists of one or more digits followed by an optional decimal point (“.”), followed optionally by one or more digits or a decimal point followed by one or more digits.

### 1.5 Polynomials

A polynomial constant for a polynomial in variable x is represented as a list of comma separated values enclosed within ‘[‘ and ‘]’. A value is a co-efficient of $x^i$ if it is at position ‘i’ in the list.

### 1.6 Strings

A string is a sequence of characters enclosed by double quotes ‘ ” ‘. A double quote inside the string is represented by ‘ \” ‘.

### 1.7 Other tokens

The following symbols or sequences of symbols are used in the language:

| | | | | | | |
|---|---|---|---|---|---|---|
| { | } | ( | ) | [ | ] | , |
| ; | + | - | * | / | % | = |
| >= | <= | > | < | == | != | : |

## 2 Types

The language is statically typed. The following types are distinguished (at compile time):

- **int** : 32-bit integers
- **float** : 32-bit IEEE floating point format
- **poly** : represents a polynomial in a single variable
- **string** : string of characters (constants only – these cannot be stored)
- **function** : user-defined functions

## 3 Expressions

### 3.1 Primary expressions

Expressions include identifiers, constants, function calls, obtaining the co-efficient of a poly variable corresponding to the i'th power, concatenation of poly variables or poly literals, obtaining the order of poly variables or literals and any other expressions surrounded by "(" and ")".

- Identifier :

  An identifier itself is a left-value expression. It will be evaluated to some value.

- Constant :

  A constant is an explicit right-value term (such as a number, polynomial, or quote-enclosed string) that will evaluate to itself in a given context.

- Function calls :

  Syntax: funcname(arg1, arg2, …)

  *funcname* is the name of the function, and *arg1*, *arg2*, etc. are the arguments (if there are any) passed to the function.

  Each argument must be an expression.

- Access to polynomial coefficients of a different power :

  Syntax: *polyname*[*i*]

  *polyname* is a polynomial identifier, and *i* is an integer.

  This evaluates to the coefficient of $x^i$ in the polynomial *polyname*.

- Concatenation of polynomials :

  Syntax: *polyA* : *polyB*

  *polyA* and *polyB* are each polynomials.

This evaluates to the polynomial formed by concatenating the coefficients list of *polyB* to the coefficient list of *polyA*. For example, "[4,3,5]:[7,6,2]" evaluates as "[4,3,5,7,6,2]".

- Order of a polynomial :

  Syntax: | *polyname* |

  *polyname* is an expression which evaluates to a polynomial

  This evaluates to the order of *polyname*, which is the highest order for which *polyname* has a non-zero coefficient.

- ( expr ) :

  The difference between *expr* and ( *expr* ) is that in the latter case, the precedence is increased (to indicate that whatever concept *expr* reduces to should be considered an atomic element). Other than that, the value is the same as *expr*.

## 3.2 Arithmetic expressions

Arithmetic expressions represent arithmetic done on some of the primary expressions. Primary expressions are taken as operands according to the following rules.

- Unary arithmetic operator "-"

  Syntax: -*a*

  *a* is an int, float, or poly.

  This returns the negative of *a*.

- Multiplicative operators

  Syntax:

  *a* / *b* (division)

  *a* % *b* (modulo)

The three operators share the same precedence level and are grouped left to right. Operands of type int and float can always be used here. If a poly is used in multiplication, it can be the first or second operand, but not both. For division, it can only be used as the first operand. The poly type can't be used in a modulo operation.

- Additive operators

  Syntax:

  *a* + *b* (addition)

  *a* - *b* (subtraction)

  *a* and *b* can both be either an int, float, or poly.

These have the same precedence level and are grouped left to right. They have a lower precedence level than the multiplicative operators.

## 3.3 Relational expressions

Syntax:

*a* >= *b* (greater than or equal to)

*a* <= *b* (less than or equal to)

*a* == *b* (equal to)

*a* > *b* (greater than)

*a* < *b* (less than)

*a* != *b* (not equal to)

*a* and *b* can each be an int, float, or poly.

In the case that the operands are some combination of ints and floats, a simple arithmetic subtraction check is made.

If the operands are polynomials, then the operators return either true or false depending on whether or not the order of the first operand can be successfully related to the order of the second operand according to the rule of the operator. If it happens that the orders are evaluated to be equal, then execution proceeds as follows:

A) In the case of an inequality operation, the coefficients of the highest order terms are compared. If they, too, are evaluated as equal, the next highest order terms are compared, and so on, until the coefficients of the lowest order terms are compared. If the coefficients of the lowest order coefficients are equal, the inequality is evaluated as false. If at any time during the traversal of the polynomials the compared numbers are evaluated as greater or less, the inequality is immediately evaluated as true or false, accordingly.

B) In the case of an equality operation, then the coefficients of the highest order terms are compared, and so on as in the case of an inequality operation. The difference is that if at any time during the traversal of the polynomials, the compared numbers are evaluated as greater or less, then the equality is immediately evaluated as false.

The values 1 or 0 (representing true and false) are returned depending on the evaluation of the operators. These operators require exactly two expressions.

## 3.4 Logical expressions

Logical operators include **not**, **and**, and **or**. They are listed in relative order of precedence, from highest to lowest. The value true is represented by 1, and false is represented by 0. Nonzero numbers, polynomials not equal to [0], and strings not equal to “” will evaluate to true. Numbers, polynomials, and strings which don’t fit these criteria will evaluate to false.

not operator

Syntax: not(*expr*)

*expr* is an expression.

If *expr* evaluates to true, not(*expr*) evaluates to false, otherwise it evaluates to true.

and operator

Syntax: *expr1* and *expr2*

*expr1* and *expr2* are both expressions

If *expr1* and *expr2* both evaluate to true, the above expression evaluates to true, and otherwise evaluates to false.

or operator

Syntax: *expr1* or *expr2*

*expr1* and *expr2* are both expressions

This evaluates to true if either *expr1* or *expr2* evaluates to true, otherwise it evaluates to false.

## 4 Statements

Statements are basically elements of a program. A sequence of statements will be executed sequentially, unless the flow-control statements indicate otherwise.

### 4.1 Groups of statements

When a sequence of consecutive statements is enclosed in braces ("{" and "}"), the statements will be treated as one entity.

### 4.2 Assignments

Assignment statements will consist of an identifier followed by an equal sign ("=") followed by an expression. The value of the expression will then be associated with the identifier.

### 4.3 Variable declarations

When an assignment statement is preceded by a variable type, the statement is interpreted as a variable declaration. Variables must be declared exactly once, and they cannot be used before they are declared. Variables must be declared at the top of a function, and at the top of the main area of a program.

### 4.4 Conditional statements

Syntax:

```
if ( logExpr )
{
  statements
}
```

or

```
if ( logExpr )
{
  statements
}
else
{
  statements2
}
```

*logExpr* is a logical expression.

*statements* and *statements2* are both lists of statements.

In an if statement, if the logical expression returns true, the first set of statements is executed, otherwise the optional second set of statements is executed. Also, any nonzero value evaluates to true, while zero evaluates to false.

## 4.5 Iterative statements

Iterative statements are performed by using a **while** loop.

Syntax:

```
while  ( logExpr  )
{
  statements
}
```

*logExpr* is a logical expression.

*statements* is a list of statements.

## 4.6 Return statements

Return statements are used to return a value from a function.

Syntax: return *val*;

*val* is the value being returned. The type of *val* must match the function's return type.

## 4.7 Exit statements

Exit statements terminate the program.

Syntax: exit;

# 5 Functions

## 5.1 Function Prototypes

Before a function is declared, it must have a prototype.

Syntax: prototype *funcname*

*funcname* is the name of the function.

## 5.2 Function Declaration

Syntax:

```
function returntype funcname (  arg1, arg2, …  )
{
  statements
}
```

*returntype* is the type of value being returned (int, poly, etc.)

*funcname* is the name of the function

*arg1*, *arg2*, etc. are the arguments (if any) which must be passed into the function.

*statements* is the optional list of statements inside the function.

The list of arguments can be empty. Functions must be all defined before the main section of the program begins. Function names can't be overloaded. Variables must be declared at the beginning of a function. Parameters are passed by value.

## 6 Internal functions

### 6.1 Print function

The print function takes one or more arguments and prints them to the standard output one by one:
print *arg1*, *arg2*, … ;

*arg1* and *arg2* are arguments to the print function.

The arguments can be variables, mathematical expressions (subtraction, addition, etc.), or quoted strings.

## 7 Program structure

### 7.1 Program Layout

The program must have prototypes first, followed by function declarations, and the main area of the program appears last. If there are no functions, the main area would be the entire program. In function declarations, as well as the main area, variables must be declared at the top.