COMS 4115 Programming Languages and Translators
PolyWiz Programming Language Proposal

Aditya Kankariya, Anthony Pitts, Max Helman, Rose Chrin, Tamjeed Azad
{ak4290, aep2195, mhh2148, crc2194, ta2553}

Spring 2021

"You're a wizard, Stephen!"

1 Introduction

PolyWiz is a strongly, statically typed and statically scoped language. It will be pass by value, not by reference, in all cases and will assume all variables are constant. The functionality of the language is intended to be similar to that of Mathematica and its primary function is to support symbolic mathematics focused on polynomial functions. In addition, it will assume basic operations from C.

2 The PolyWiz Language

2.1 Data Types

The language supports primitive data types of int, float, and booleans, and additionally supports type string and array. Using floats and arrays as building blocks, the language fundamentally supports a new type named poly. Additional types will be defined as needed.

2.1.1 Type: int

The int type represents numerical integers and takes up 4 bytes of memory. Syntax and operations are mostly C-like, specifically supporting the operations +,-,*,/,%,=,+=,-=,*=,/=. Additionally, int supports the boolean operations and,or,<,>,<=,>=.

2.1.2 Type: boolean

The boolean type can only have two values, true or false. This takes up 4 bytes of memory and supports the boolean operations and,or,<,>,<=,>=. Implementation could simply be an int of value 0 or 1 for false and true, respectively under the hood, but other implementations are also possible.

2.1.3 Type: float

This float type represents floating point numbers, used to approximate non-integer real numbers. Using 8 bytes of space, implemented using IEEE 754-1985 double precision standard, it can precisely approximate real numbers in the range of $\pm 2.23 \times 10^{-308}$ to $\pm 1.80 \times 10^{308}$. It supports all operations that the int type supports.

2.1.4 Type: string

The string type represents a concatenated, immutable block of either ASCII or utf-8 characters (which of these will be the final native character set will be decided soon). It uses memory dynamically based on the size of string, and generally supports operations supported by Python's str type, including concatenation, indexing and reversal. It is important to note that this language does not support the char type common to languages such as C and Java; all single chars are of type string with length 1.

2.1.5 Type: array

The array type is specified using [*type*], and does not have mutable length and uses C-like syntax. Arrays can only consist of a single type and are immutable.

2.1.6 Type: poly

This is the central, defining type of the PolyWiz language. The poly type specifies polynomials using an array of floats as variable constants and an array of floats for exponents, used in tandem to define polynomial functions. Polynomials can only be defined in terms of a single variable, say x, and they are instantiated in the following way:

```
// poly example
poly polynomial1 = new_poly([1.0, 2.0, 4.0], [3.0, 2.0, 1.0]); // polynomial1 = x^3 + 2x^2 + 4x
```

This would represent the instantiation of the polynomial $1.0x^{3.0} + 2.0x^{2.0} + 4.0x^{1.0}$.
Formulating polynomials in this fashion allow for swift, intuitive support for operations common in polynomial focused tasks, such as addition, multiplication, plotting, and composition of polynomials. This is the heart of this language; the other types primarily serve to make this type operate with ease. Operations supported by this poly type are detailed below.

2.2 Polynomial Operations

2.2.1 Polynomial Addition, +

Return Type: poly
Operands: Two poly variables on both sides of the addition operator (+)
Operation Logic: Returns the polynomial sum of the two polynomial operands.

```
    // + operator example
    poly poly1 = new_poly([2.0, 4.0, 2.0], [2.0, 1.0, 0.0]); // poly1 = 2x^2 + 4x + 2
    poly poly2 = new_poly([1.0, 2.0, 3.0], [2.0, 1.0, 0.0]); // poly2 = x^2 + 2x + 3
    poly poly_sum = poly1 + poly2; // poly_sum = 3x^2 + 6x + 5
```

2.2.2 Polynomial Subtraction, -

Return Type: poly
Operands: Two Poly variables on both sides of the subtraction operator (-)
Operation Logic: Returns the polynomial difference of the two polynomial operands.

```
    // - operator example
    poly poly1 = new_poly([2.0, 4.0, 2.0], [2.0, 1.0, 0.0]); // poly1 = 2x^2 + 4x + 2
    poly poly2 = new_poly([1.0, 2.0, 3.0], [2.0, 1.0, 0.0]); // poly2 = x^2 + 2x + 3
    poly poly_difference = poly1 - poly2; // poly_difference = x^2 + 2x - 1
```

2.2.3 Polynomial Multiplication, *

Return Type: poly
Operands: Two Poly variables on both sides of the multiplication operator (*)
Operation Logic: Returns the two polynomial operands' product.

```
    // * operator example
    poly poly1 = new_poly([1.0], [1.0]); // poly1 = x
    poly poly2 = new_poly([1.0, 2.0, 3.0], [2.0, 1.0, 0.0]); // poly2 = x^2 + 2x + 3
    poly poly_product = poly1 * poly2; // poly_product = x^3 + 2x^2 + 3x
```

2.2.4 Polynomial Division, /

Return Type: poly
Operands: Two Poly variables on both sides of the division operator (/)
Operation Logic: Returns the first polynomial divided by the second polynomial.

```
    // / operator example
    poly poly1 = new_poly([1.0], [2.0]); // poly1 = x^2
    poly poly2 = new_poly([1.0], [1.0]); // poly2 = x
    poly poly_div = poly1 / poly2; // poly_div = x
```

2.2.5 Constants Retriever,

Return Type: [float]
Operand: Poly variable on left side of the operator (#)
Operation Logic: Returns an array of the polynomial constants, from highest to lowest order.

```
    // # operator example
    poly poly1 = new_poly([1.0, 2.0, 3.0], [2.0, 1.0, 0.0]); // poly1 = x^2 + 2x + 3
    [float] poly1_constants = poly1# ; // poly1_constants = [1.0, 2.0, 3.0]
```

2.2.6 Polynomial Composition, @

Return Type: poly or float/int
Operands: Two Poly variables or one Poly and one float/int variable on the left and right side, respectively, of the composition operator (@)
Operation Logic: Returns the value of the polynomial on the left hand side of the operator, composed with the float/int or second polynomial, on the right hand side.

```
    // @ operator example, composing a poly with another poly
    poly poly1 = new_poly([1.0], [2.0]); // poly1 = x^2
    poly poly2 = new_poly([1.0], [2.0]); // poly2 = x^2
    poly poly_composed = poly1 @ poly2; //poly_composed = x^4

    // @ operator example, evaluating a poly at a specified independent variable location
    poly poly1 = new_poly([1.0], [2.0]); // poly1 = x^2
    float poly1_value = poly1 @ 2; // poly1_val = 4.0
```

2.2.7 Convert Polynomial to String, to_str

Return Type: string
Operand: Poly variable on left side of the operator (print)
Operation Logic: Returns a string representation of the polynomial.

```
// # operator example
poly poly1 = new_poly([1.0, 2.0, 3.0], [2.0, 1.0, 0.0]); // poly1 = x^2 + 2x + 3
string poly1_str = to_str poly1; // poly1_str = "x^2+2x+3"
```

2.2.8 Power, ˆ

Return Type: float/int
Operand: Two floats/ints on each side of the power operator (ˆ)
Operation Logic: Returns the left hand side float/int raised to the right hand side float/int.

```
// ^ operator example
float power_result = 2.0 ^ 3.0; // power_result = 8.0
```

2.2.9 Absolute Value, ||

Return Type: float/int
Operand: A float/int inside the absolute value operator bars (||)
Operation Logic: Returns the absolute value of the float/int inside the absolute value bars.

```
// | | operator example
float abs_value_result = |-2.0|; // abs_value_result = 2.0
```

2.3 Keywords

The reserved keywords in PolyWiz are:
in, range, for, length, print, to_str, plot, and, or, not, def, true, false, return, if, else, elif, void
We also reserve // for single-line comments, and /* and */ are reserved for multi-line comments.

3 Control Flow

3.1 If-Elif-Else Statements

The if-elif-else statements are used to make decisions based on the expression being evaluated. Any expression must evaluate to a valid boolean in order to compile.

```
// Example of if/elif/else control flow
int x = 5;
if x > 100 { // False, so program moves onto elif statement
   print("x is greater than 100");
}
elif x > 10 { // False, so program moves onto else statement
   print("x is greater than 10");
}
else {
   print("x is less than or equal to 10"); // This is what the program outputs
}
```

3.2 For Loops

The for statement iterates over the items of any sequence (an array or a string), in the order that they appear in the sequence. In order to iterate over a numerical incrementor, you can use the range() function.

```
// To print out all the items in an array
for element in my_array {
    print(element);
}

// To print out every even integer between 0 and 9
// Default start parameter in range is 0, default step parameter is 1
for int i in range(0, 10, 2) {
    print(i);
}

// To iterate over the indices of a sequence, you can combine range() and len()
for int i in range(len(my_array)) {
    print(my_array[i]);
}
```

4 Function Declaration

PolyWiz supports function declarations. If the function returns a value, you must declare the return type in the method signature. If the function has no return value, then you can declare *void* in place of the return type.
Functions are first-class objects and can be passed by name to other functions.

```
// This function adds 5 to any integer value passed into it and returns the resulting integer
def int add_five(int x) {
    return x + 5;
}

// This function takes in another existing function f which modifies any integer value in some mysterious way
def void modify([int] my_array, f) {
    for int i in range(len(my_array)) {
        my_array[i] = f(my_array[i]);
    }
}
```

5 Standard Library Functions

5.1 new_poly([float] *coefficients*, [float] *exponents*)

Create a new polynomial of the type *poly*. The first argument is a single list of coefficients of type *float* for each term. The second argument is a single list of exponents of type *float* for each respective term. Both lists should align and be of the same length, meaning that the i'th value in both *coefficients* and *exponents* correspond to the same term. This function is linked to the C standard library under the hood in order to create a variable of type *Poly*, which is represented with some form of a dictionary.

```
// Example of new_poly
poly poly1 = new_poly([1.0], [1.0]); // poly1 = x
poly poly2 = new_poly([1.0, 2.0, 3.0], [2.0, 1.0, 0.0]); // poly2 = x^2 + 2x + 3
```

5.2 *p*.order()

Return an integer containing the order/degree of polynomial p of type *Poly*.

```
    // Example of order
    poly poly1 = new_poly([1.0, 2.0, 3.0], [2.0, 1.0, 0.0]); // poly1 = x^2 + 2x + 3
    int poly1_order = poly1.order(); // poly1_order = 2
```

5.3 TEX Integration

As the old adage goes, if it's in LATEX, it must be true. Since the PolyWiz language focuses on symbolic mathematics and LATEX is the most popular method of typesetting mathematics, PolyWiz will include a function that outputs the typeset polynomial to provide seamless TEX integration.

5.3.1 TEX formatting, print_tex

Return Type: string
Operand: Poly variable on left side of the operator (print_tex)
Function Logic: Returns a typeset string representation of the polynomial.

```
    // # operator example
    poly poly1 = new_poly([1.0, 2.0, 3.0], [2.0, 1.0, 0.0]); // poly1 = x^2 + 2x + 3
    string poly1_str = poly1.print_tex() ; // poly1_str = "$$x^{2}+2x+3$$"
```

5.4 Graphical Output

PolyWiz will support basic plotting of 2D polynomial functions, allowing for customization of both x and y range. If no ranges are given, a default range will be chosen. Plot will produce a basic plot which will be output to the filepath passed as an argument.

```
    //graph example
    poly poly1 = new_poly([1.0, 2.0, 3.0], [2.0, 1.0, 0.0]); // poly1 = x^2 + 2x + 3
    //plot(<FILEPATH>, [list of polynomials], x min, x max, y min, y max)
    plot(<FILEPATH>, [poly1], -10, 10, 0, 20);
```

6 Example

Here is a simple function that composes two polynomials, plots the composed polynomials, and then returns the composition formatted in LATEX:

```
def string compose_n_graph(float a, float b) {
    poly poly1 = new_poly([a, 2.0, 3.0], [2.0, 1.0, 0.0]); // poly1 = ax^2 + 2x + 3
    poly poly2 = new_poly([2.0, 4.0, 6.0], [b, 2.0, 3.0]); // poly2 = 2x^b + 4x^2 + 6x^3
    poly poly3 = poly2 @ poly1; //composes the two polynomials
    for int i in range(-10, 10, 2) {
        float tmpy1 = poly1 @ i;
        float tmpy2 = poly2 @ i;
        float tmpy3 = poly3 @ i;
        print(i, tmpy1, tmpy2, tmpy3);
    }
    plot("zaphod2/~/edwards/desktop/pretty_polynomial.png", [poly1, poly2, poly3], -20, 20, -20, 20);

    string nice_n_tex = poly3.print_tex();
    return nice_n_tex;
}
compose_n_graph(2.0,2.0);
```