COMS 4115 Programming Languages and Translators
PolyWiz Language Reference Manual

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

Spring 2021

Contents

1 Overview of PolyWiz

1.1 A Mathematician's Dream

PolyWiz is truly a mathematician's dream. The functionality of the language is intended to be similar to that of Mathematica. The primary goal is to support symbolic mathematics focused on polynomial functions. In addition, PolyWiz aims to provide unique TEX integration, allowing a user to not only perform numerical calculations but produce ready-to-show documents with plots.

1.2 General Language Features

PolyWiz is a strongly, statically typed and statically scoped language. It will be pass by reference behind the scenes to the programmer. This will enhance memory in the long run given all of the types in the language are immutable. In addition, PolyWiz will assume that all variables are constant. The language will have syntax most similar to C, with some syntax having close similarity to Python as well. It will also assume all basic operations from C.

2 Basic Syntax

2.1 Comments

All comments are multiline, beginning with "/*" and ending with "*/".

```
/* PolyWiz is an incredible
programming language for
symbolic mathematics and
other fun things */

/* OK let's write some code now */
string my_str = "I like PolyWiz"; /* But in reality, I love PolyWiz */
```

The grammar parser never sees comments that are made in the code, as the scanner ensures that their contents are never tokenized.

2.2 Identifiers

Identifiers in PolyWiz denote variable and functions. All identifiers consist of ASCII letters (non case-sensitive) and decimal digits; no special characters are permitted other than underscores. The first character must be an ASCII letter, not a number or an underscore. Identifiers cannot be reserved keywords. Here are some permitted identifiers:

```
edwards_rocks, EdwardsRocks, plt_rocks, length_poly, cs_makes_me_1000000dollars
```

Here are examples of identifiers that are not allowed:

```
4115_rocks, length, cs_makes_me_$1000000
```

In the grammar parser, these are denoted by the token `ID`.

2.3 Reserved Keywords

Keywords in PolyWiz are reserved identifiers that cannot be used for any other purpose. PolyWiz has the following keywords:

```
for, length, and, or, not, return, if, else, elif, def, void, print, tex_document,
plot, int, string, float, array, poly, pass, continue, break, in
```

In the grammar parser, these are all denoted by capitalized letters of their words, for example `NOT, IF, POLY`.

2.4 Braces

PolyWiz uses braces to group statements and enforce static scoping. Whitespace (specifically empty lines, tabs, and extra spaces) is ignored. Control flow keywords must be followed by braces. Here is some sample code that demonstrates the proper use of braces:

```
/* Pretty code that works */
if (i > 27){
print("i is greater than 27");
}
```

Meanwhile, this code is a poor stylistic choice but is equivalent to the previous code:

```
/* Ugly code that works */
if         (i        > 27
)                 {
print(

"i is greater than 27");}
```

This code is not correct because it does not have the requisite braces after the if statement:

```
/* Pretty code that does not work */
if (i > 27)
   print("i is greater than 27");
```

In the parser, these are denoted by `LBRACE` and `RBRACE`.

2.5 Parentheses

Parentheses are used to override default precedence; anything inside parentheses automatically becomes the highest precedence.

```
/* Parentheses example */
int a = 1 + 2 * 3; /* a = 7 */
int b = (1 + 2) * 3; /* b = 9 */
```

2.6 Sequencing, ;

Return Type: $< T >$ where T is type of RHS expression's return value
Operand: Two expressions on each side of ; operator
Operation Logic: Evaluates the LHS expression, followed by the RHS expression.

```
/* ; operator example */
a = 5; 6 /* returns the value 6 after assigning a = 5 */
```

In the parser, these are denoted by `LPAREN` and `RPAREN`.

3 Data Types and Literals

3.1 Mutability

All data types in PolyWiz are inherently immutable; rather, variable assignment and reassignment are supported. PolyWiz is pass-by-reference.

3.2 Literals

The language supports literals of type int, boolean, float, and string. See data types section for details of these.

The int literal is represented as a sequence of digits from the set [0,9], and is representative of the int data type.

Representative regular expression: ['0'-'9']+

The string literal is represented as a sequence of ASCII characters, not starting with a number but can include numbers. It is representative of the string data type.

Representative regular expression: [string regex]

The float is represented as a sequence of digits, with a single decimal point within the sequence body, with this sequence of digits possibly raised to some exponent. It is representative of the float data type.

Representative regular expression: [float regex]

The boolean literal is represented by the keywords 'True' and 'False'; it represents the boolean data type.

Representative regular expression: [boolean regex]

3.3 Primitive 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.

Primitive data types are all immutable in this language. When say, a variable is assigned one of these type values and it is changed, a completely new value is assigned and the old one is discarded.

3.4 Booleans

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.

Examples:

```
boolean x = True;
if (not x) {
  print("x is not true");
}
```

3.4.1 Ints

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,<,>,<=,>=.

Examples:

```
/* examples of ints: 4, 23, -5623 */
int x = 57;
x = x + 9; /* integer addition */
```

3.4.2 Floats

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.

The support for very large and very small numbers eliminates the need for types such as long and short. When these boundaries are exceeded, the language simply returns an overflow error.

These values use the same symbols for operations as the int type. Operations can occur between ints and floats, which would return a float. See operators section for more details.

Examples:

```
/* examples of float: -8.4, 71234.98, 1.234 * 10^59 */
float x = 4.0;
x = x * 3.5; /* multiplying floats */
```

3.4.3 Strings

The string type represents a concatenated, immutable block of either ASCII characters. 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. They are declared and specified in the language using either double quotes or single quotes.

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.

```
string x = "stephen";
x = x + " " + "edwards" /* concatenation of strings */
```

3.5 Arrays

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. Only 1D arrays are supported.

```

[ float ] x = [ 4.5, 9.6, 3.2, 4.9 ];

for number in x {
  print(number);
}
```

3.6 The Poly Data Type

The poly data type is the centerpiece of the PolyWiz language. Much of Polywiz's standard library and operations, such as addition (+), multiplication (*), composition (@), plotting (plot()), are built around functionality with the type poly. The poly type specifies polynomials using an array of floats as variable coefficients and an array of floats for variable 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:

3.6.1 Implementation

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, 2.0, 4.0], [3.0, 2.0, 1.0]); /* poly1 = x^3 + 2x^2 + 4x */
    poly poly2 = new_poly([1.0], [1.0]); /* poly2 = x */
```

This instantiates the polynomials $1.0x^{3.0} + 2.0x^{2.0} + 4.0x^{1.0}$ and $1.0x^{1.0}$.

3.6.2 *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 */
```

3.7 Data Types in PolyWiz Grammar

Each of float, int, boolean, string, are all tokenized as literals of their type in the grammar parser (`BLIT, FLIT, SLIT`), and they each also have a separate token for their types, such as `FLOAT, INT, BOOL`. The poly type has a `POLY` token for its type, but its 'literal' is handled differently as instantiation is handled via the new_poly function, and is generalized under `expr`. All types of arrays have their own literal token, such as `FLOAT_ARR_LIT`, `STRING_ARR_LIT` and a token for their specific types, such as `FLOAT_ARR`.

4 Statements and Expressions

4.1 Statements

A PolyWiz program is made up of a combination of the following types of statements:
Expressions, Variable assignments, Return statements, Function definitions, Function calls, If-elif-else statements, For loops

4.1.1 If-Elif-Else Statements

If-elif-else statements are used to make decisions based on the expression (condition) being evaluated. If a condition evaluates to true, the statements inside the if statement are evaluated, otherwise the program will either move on or evaluate an optional else statement as shown below. Any expression must evaluate to a valid boolean (true/false) in order to compile. The condition must be wrapped in parenthesis.

```
/* 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 */
}
```

4.1.2 For Loops

For loops in PolyWiz are incredibly similar to C: they are contained in parenthesis, and consist of a variable initialized to some initial variable followed by a semicolon, then a breaking condition followed by a semicolon, and finally an update rule for the variable. Use the *break* keyword to exit out of a loop and use the *continue* keyword to return control to the beginning of the loop for the next item in the sequence. You can modify all the variables in an array by iterating over the indices of the array.

```
/* To print out all the items in an array of integers until we see an integer greater than 10 */
for (int i = 0; i < length(my_array); i++) {
   if(my_array[i] > 10) {
      break; /* Saw an integer greater than 10, exits loop */
   }
   print(my_array[i]);
   element = 0; /* This will not affect my_array */
}

/* To print out every even integer between 0 and 9 */
for (int i = 1; i < 10; i = i + 2) {
   print(i);
}

/* To iterate over the indices of a sequence, you can combine a for loop with length() */
for (int i = 0; i < length(my_array); i++) {
   print(my_array[i]);
}
```

4.2 Expressions and Operators

Our language supports elementary unary and binary operators to accomplish a plethora of tasks.

4.2.1 Unary Operators

In the parser, this is represented by
| Unop_right expr * operation * operation
and by
| Unop_left operation * expr There is only one operation whose operation is to the left of the expression, and that is the logical not operation, denoted by the keyword 'not'.

4.2.2 Binary Operators

In the parser, this is generally represented by

| Binop expr * operation * expr

There are several operators that evaluate via these scheme.

Binary arithmetic operators are +,-,*,/,and %. These all directly return a new expression literal based on expression type, usually ints or floats; a notable exception is the string type, which uses '+' for concatenation.

We have logical operators that can be nested to represent complex boolean expressions; these logical operators are the logical $\wedge$ and $\vee$ operations, and and or.

We have comparison operations (essential for things like control flow) that always return booleans. These are ==, ! =, >, <, >=, and <=.

For assignment, the parser specifically represents this by

| Assign var * operation * expr

Binary assignment operators are used to assign values to variables.

4.3 Operations

4.3.1 Addition, +

Return Type: $< T >$ where T is the type of both operands
Operands: Two variables of type $< T >$ on both sides of the addition operator (+)
Operation Logic: Returns the sum, which could be the polynomial sum, of the two operands.
Grammar:

```
%left PLUS
expr:
        expr PLUS expr { Binop($1, Add, $3) }
```

```
/* + 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 */
```

4.3.2 Subtraction, -

Return Type: $< T >$ where T is the type of all operands
Operands: One or Two variables of type $< T >$, with at least one variable on the RHS of the subtraction operator (-)
Operation Logic: Returns the difference, which could be of polynomials, of the two operands.
If only one operand is supplied, it returns -1.0 * operand.
Grammar:

```
%left MINUS
expr:
        expr MINUS expr { Binop($1, Sub, $3) }
```

```
/* - 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 */
```

4.3.3 Multiplication, *

Return Type: $< T >$ where T is the type of both operands
Operands: Two variables of type $< T >$ on both sides of the multiplication operator (*)
Operation Logic: Returns the two operands' product, which could be polynomial multiplication.
Grammar:

```
%left TIMES
expr:
        expr TIMES expr { Binop($1, Mult, $3) }
```

```
/* * 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 */
```

4.3.4 Division, /

Return Type: $< T >$ where T is the type of both operands
Operands: Two variables of type $< T >$ on both sides of the division operator (/)
Operation Logic: Returns the first operand divided by the second, which could be polynomial division.
Grammar:

```
%left DIVIDE
expr:
    expr DIVIDE expr { Binop($1, Div, $3) }
```

```
    /* / 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 */
```

4.3.5 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.
Grammar:

```
%left CONST_RETRIEVER
expr:
    expr CONST_RETRIEEVER { Unop(Const_retriever, $1) }
```

```
    /* # 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] */
```

4.3.6 Polynomial Composition, :

Return Type: poly
Operands: Two poly variables on the left and right side of the composition operator (:)
Operation Logic: Returns the polynomial that forms by composing the polynomial on the left hand side of the : operator with the second polynomial, on the right hand side.
Grammar:

```
%left COMP_POLY
expr:
    expr COMP_POLY expr { Binop($1, Comp_poly, $3) }
```

```
    /* : 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^2)^2 = x^4 */
```

4.3.7 Polynomial Evaluation, @

Return Type: float
Operands: One Poly and one float/int variable on the left and right side, respectively, of the @ operator
Operation Logic: Returns the value of the polynomial at the float/int location specified.
Grammar:

```
%left EVAL_POLY
```

expr:
  expr EVAL_POLY expr { Binop($1, Eval_poly, $3) }

```
/* @ 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 */
```

4.3.8 Convert Polynomial to String, to_str

Return Type: string
Operand: Poly variable
Operation Logic: Returns a string representation of the polynomial.
Grammar:

%left TO_STR
expr:
  TO_STR LPAREN expr RPAREN { Make_str($3) }

```
/* # 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" */
```

4.3.9 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.
Grammar:

%left EXP
expr:
  expr EXP expr { Binop($1, Exp, $3) }

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

4.3.10 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.
Grammar:

%left ABS_VALUE
expr:
  ABS_VALUE expr ABS_VALUE { Unop(Abs_value, $1) }

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

4.3.11 Assignment, =

Return Type: void
Operand: A string on the LHS and a type $< T >$ on the RHS of the = operator
Operation Logic: If the RHS is a primitive, it stores the RHS' value into a variable, named the LHS string value. Otherwise, it stores the RHS' pointer location into a variable, named the LHS string value
Grammar:

```
%right ASSIGN
expr:
    ID ASSIGN expr { Assign($1, $3) }
```

```
/* = operator example */
a = 5; /* Stores value 5 in variable "a" */

poly poly1 = new_poly([1.0, 2.0, 3.0], [2.0, 1.0, 0.0]);
poly poly2 = poly1; /* poly2 holds a pointer to poly1 */
```

4.3.12 Boolean Negation, not

Return Type: boolean
Operand: A boolean on the RHS of the not operator
Operation Logic: Returns the opposite boolean value as the operand.
Grammar:

```
%right NOT
expr:
    NOT expr { Unop(Not, $1) }
```

```
/* not operator example */
a = not 1==1 ; /* a = false */
```

4.3.13 Equality Comparison, ==

Return Type: boolean
Operand: Two values of type $< T >$ on each side of the == operator
Operation Logic: Returns True if both operands are of equal value.
Grammar:

```
%left EQ
expr:
    expr EQ expr { Binop($1, Equal, $3) }
```

```
/* == operator example */
poly poly1 = new_poly([1.0, 2.0, 3.0], [2.0, 1.0, 0.0]);
poly poly2 = poly1;
a = poly1 == poly2; /* a = true */
```

4.3.14 Less than comparison, <

Return Type: boolean
Operand: Two values of type $< T >$ on each side of the < operator
Operation Logic: Returns True if LHS is strictly less than RHS.
Grammar:

```
%left LT
expr:
    expr LT expr { Binop($1, Less, $3) }
```

```
/* < operator example */
a = 1 < 1; /* a = false */
```

4.3.15 Less than or equal to comparison, <=

Return Type: boolean
Operand: Two values of type $< T >$ on each side of the <= operator

Operation Logic: Returns True if LHS is less than or equal to RHS.
Grammar:

```
%left LEQ
expr:
      expr LEQ expr { Binop($1, Leq, $3) }
```

```
/* <= operator example */
a = 1 <= 1; /* a = true */
```

4.3.16 Greater than comparison, >

Return Type: boolean
Operand: Two values of type $< T >$ on each side of the > operator
Operation Logic: Returns True if LHS is strictly greater than RHS.
Grammar:

```
%left GT
expr:
      expr GT expr { Binop($1, Greater, $3) }
```

```
/* > operator example */
a = 1 > 1; /* a = false */
```

4.3.17 Greater than or equal to comparison, >=

Return Type: boolean
Operand: Two values of type $< T >$ on each side of the >= operator
Operation Logic: Returns True if LHS is greater than or equal to RHS.
Grammar:

```
%left GEQ
expr:
      expr GEQ expr { Binop($1, Geq, $3) }
```

```
/* >= operator example */
a = 1 >= 1; /* a = true */
```

4.3.18 Boolean or, *or*

Return Type: boolean
Operand: Two boolean values on each side of the or operator
Operation Logic: Returns True if LHS or RHS is true.
Grammar:

```
%left OR
expr:
      expr OR expr { Binop($1, Or, $3) }
```

```
/* or operator example */
a = true or false; /* a = true */
```

4.3.19 Boolean and, *and*

Return Type: boolean
Operand: Two boolean values on each side of the or operator
Operation Logic: Returns True if both the LHS and RHS is true.
Grammar:

```
%left AND
expr:
        expr AND expr { Binop($1, And, $3) }
```

```
/* and operator example */
a = true and false; /* a = false */
```

4.3.20 Membership, in

Return Type: boolean
Operand: A value on the left-hand side and an array or string on the right-hand side
Operation Logic: Returns true if the specified value is a member of the array or a substring of the string; false otherwise
Grammar:

```
%left IN
expr:
        expr IN expr { Binop($1, InArray, $3) }
```

```
/* in example */
[int] arr = [5,6,7,8,9];
print(4 in arr); /* prints false */
print(5 in arr); /* prints true */

string s = "edwards";
print("ed" in s); /* prints true */
print("eddy" in s); /* prints false */
```

4.4 Operator Precedence

This operator precedence table specifies, in increasing order, the compiler's priority and associativity for each operator.

Operator	Meaning	Associativity
;	Sequencing	Left to Right
=	Assignment	Right to Left
not	Boolean Negation	Right to Left
==, >, <, >=, <=	Comparisons	Left to Right
or	Or	Left to Right
and	And	Left to Right
ˆ	Power	Left to Right
||	Absolute Value	Non-associative
to_str	Convert poly to string	Left to Right
+, -	Addition, Subtraction	Left to Right
*, /	Multiplication, Division	Left to Right
-	Unary Subtraction	Non-associative
#	Constants Retriever	Left to Right
@	Evaluation	Left to Right
:	Composition	Left to Right

4.5 Functions

In the language a function is a statement that will take a list of arguments and return a single value or nothing. The list of arguments that it takes in will require type specification. The function definition will start with the keyword def and then the return type. If it returns nothing, keyword **void** is used instead. Its identifier will follow the return type.
Grammar:

```
fdecl:
```

DEF typ ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE

```
/*function definition example */
def string tex_string(float a, float b) { /* { begins body */
   poly poly1 = new_poly([a, 2.0, 3.0], [2.0, b, 0.0]);
   string nice_n_tex = poly1.print_tex();
   return nice_n_tex; /*return statement with nice_n_tex type string
} /* closes body */

/*returns nothing */
def void add_poly(float a, float b) {
   poly poly1 = new_poly([a, 2.0, 3.0], [2.0, b, 0.0]);
   poly poly2 = new_poly([a, 2.0, 3.0], [2.0, b, 0.0]);
   poly poly3 = poly1 + poly2;
}
```

4.6 Function Calls

To call a function, the identifier along with its arguments in parentheses will be used. If a function is called using improper types or without sufficient arguments, an error will be raised during compilation, depending on why the arguments failed. In the grammar, a function call is an expression (expr), so it can be assigned to a variable or stand on its own.

```
/*function call examples */
tex_string(2.0, 2.0); /*outputs a string */
tex_string(5.0, 2.0); /*outputs a string */
tex_string(5.0); /*would raise an error at compile time */
tex_string(5, 2); /*would raise a TypeError at runtime */
```

4.6.1 Variable Assignment from Functions

Variables can be assigned to the return value of a function assuming the return type of the function and the type of the variable are the same. If they are not, this will raise a TypeError at runtime. In the grammar, this is done as an expression (expr) and is given as expression `EQ` expression.

```
/*variable assignment examples */
string poly_tex_one = tex_string(2.0, 2.0); /*outputs a string */
int poly_tex_one = tex_string(2.0, 2.0); /*would raise a TypeError at runtime */
```

5 Standard Library

6 Standard Library

6.1 Sequence operations

PolyWiz supports the following operations with both arrays and strings:

Method/operation	**Result**
str[i]	Returns the i'th item in str Raises an IndexError if i is outside the range of str
str[i:j]	Returns the substring/array between the i'th item and the j'th item in str If i is not defined, the substring/array from index 0 up to j is returned, and vice versa if j is not defined Raises an IndexError if i or j is outside the range of str
str1 + str2	Returns a concatenation of both str1 and str2 in the form of a single string or array
length(str)	Returns the size/length of str in terms of its number of characters or items

Grammar:

expr :

```
        array_operation { $1 }
    array_operation:
        expr LBRACK expr RBRACK { IndexArray($1, $3) }
      | expr LBRACK expr COLON expr RBRACK { SliceArray($1, $3, $5) }
      | LENGTH LPAREN expr RPAREN { Length($3) }
```

6.2 Printing

PolyWiz supports using print() to display a string representation of any built-in type in standard output. print() cannot take in a concatenation of two different types, it will raise a TypeError.

Method: print()
Return Type: string
Parameter: Any expression or variable of a built-in type
Function Logic: Outputs a string to stdout representing function input.
Grammar:

```
    expr:
        PRINT LPAREN expr RPAREN { Print_stuff($3) }
```

```
/*print example */
string a = "Hello";
print(a); /* Standard output will display: Hello */
print(7); /* Standard output will display: 7 */
print(a + 7); /* This will raise a TypeError, cannot concatenate a string and an int */
```

6.3 Plotting

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.

Method: plot()
Return Type: string
Parameters: filepath, a list of polynomials, x min, x max, y min, y max
Function Logic: Returns a string that is the filepath where the plot was produced.
Grammar:

```
    expr:
        PLOT LPAREN expr COMMA POLY_ARR_LIT COMMA expr COMMA expr COMMA expr
            COMMA expr RPAREN { Plot_stuff($1, $3, $5, $7, $9, $11, $13) }
```

```
/*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);
```

6.4 TEX Integration

LATEX is the true mathematician's language, and as such, PolyWiz is designed to support seamless TEX integration. Every poly can be formatted in TEX , and entire documents including plots can be generated easily.

6.4.1 TEX Formatting

Method: print_poly()
Return Type: string
Operand: Poly variable on left side of the method (print_tex)

Function Logic: Returns a string representation of the typeset polynomial.
Grammar:

```
expr:
    PRINT_TEX LPAREN expr RPAREN { Print_latex($3) }
```

```
/* print_poly() 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 = "$$x^{2}+2x+3$$" */
```

6.4.2 Generating TEX Documents

Function: tex_document()
Return Type: string
Parameters: array *a* of strings containing text, typeset equations, and file paths to saved plots, array *b* of indices of plots
Function Logic: Returns TEX document containing equations and plots in the format of a string
Grammar:

```
expr:
    TEX_DOC LPAREN STRING_ARR_LIT COMMA INT_ARR_LIT RPAREN
        { Generate_doc($3, $5) }
```

```
/* tex_document() example */
poly poly1 = new_poly([1.0, 2.0, 3.0], [2.0, 1.0, 0.0]); /* poly1 = x^2 + 2x + 3 */
string intro = "After much research, we present the Edwards polynomial:";
string outro = "This will revolutionize the field of compilers.";
string poly1_str = poly1.print_tex() ; /* poly1_str = "$$x^{2}+2x+3$$" */
string poly1_plt = plot("mypc/coms4115/edwards.png", [poly1], -10, 10, 0, 20);
string doc1 = tex_document([intro, poly1_plt, poyl1_str, outro], [1]); /* generate document */
print(doc1); /* print document (to std out) */
```

This prints the following text:

```
\documentclass{article}
\begin{document}
\usepackage{graphicx}
After much research, we present the Edwards polynomial:
\begin{figure}[!h]
\centering
\includegraphics[width=3.5in]{mypc/coms4115/edwards.png}
\label{fig_sim}
\end{figure}
$$x^{2}+2x+3$$
This will revolutionize the field of compilers.
\end{document}
```

6.5 Exception Handling

PolyWiz will support basic handling of errors at runtime by allowing you to wrap a try-catch-finally mechanism around a block of code. This functionality can be very useful when a small chunk of code or statement in a program inputs arguments of a wrong type into a function for example, and even if that statement does not run properly, it does not hinder the flow of the rest of the program.

Grammar:

```
stmt:
    TRY LBRACE stmt_list RBRACE catch NOFINALLY { TryCatch(List.rev $3, $5, []) }
  | TRY LBRACE stmt_list RBRACE catch FINALLY LBRACE stmt_list RBRACE
        { TryCatch(List.rev $3, $5, List.rev $8) }
catch:
    CATCH LPAREN expr RPAREN LBRACE stmt_list RBRACE { [Catch($3, List.rev $6)] }
```

| CATCH LPAREN expr RPAREN LBRACE stmt_list RBRACE catch { Catch($3, List.rev $6) :: $8 }

```
// example of exception handling
void print_twelfth([int] arr) {
// Attempting to index and print out the 12th item in the array would raise an IndexError, thus we allow the
     program to "catch" this error and continue on
    try {
        print("Twelfth item: " + to_str(arr[11]));
    }
    catch(IndexError) {
    /* This will display in standard output */
        print("Array contains less than 12 items.");
    }
    finally {
    /* This will also display in standard output regardless of the try-catch evaluation */
        print("Finally we can go to Catch!");
    }
}

[int] arr = [1, 4, 6, 3, 0];
print_twelfth(arr);
print("Let's hit Catch!")
```

7 Sample Code

Here is a simple function that composes two polynomials, plots the original and 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 = 1; i < 50; i = i + 1) {
        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);
```