Polynomial Interpreter

Bogdan Morosanu
30421

1. General Objectives & Features :
Implementing a simple polynomial script interpreting environment based on regex String pattern matching that is capable of running arbitrary scripts. The Polynomial class is designed to be functional in character. That is, it avoids mutable state. All operations on polynomials generate new polynomials rather than mutate existing ones in place.

Variables:
Our polynomial interpreting environment should contain two data types that can be instantiated: integer arrays and polynomials with integer coefficients of one input variable. Our environment should also support variable reassignment and namespace clearing.

Literals:
The interpreting environment should also have support for integer literals, integer array literals and polynomial literals. integer literals however, can only be passed to and from polynomial arguments, instead of being instantiated alone. Besides it should also support array generation using both enumerations and a generator-style syntax which specifies range start, end and an optional increment step.

Operations:
Operations contained should be polynomial evaluation (both at single integer points and over whole arrays), polynomial generation using arithmetic (polynomial addition, polynomial subtraction, polynomial multiplication, polynomial division and polynomial remainders) along with polynomial generation using calculus operators (integrals and derivatives).

Functional Character:
The Polynomial class is designed to be functional in character. That is, it avoids mutable state. Once created, there is no way to alter the underlying coefficient array. All operations on polynomials generate new polynomials rather than mutate existing ones in place. This provides for easier reasoning and development of the algorithms, since we can apply any operation to the existing polynomials without fear of affecting other parts of our program that might have a reference cached to the Polynomial object that we are manipulating. More specifically there is no need for a .clone() method and reference identity is the same thing as value identity (though more polynomials can be created with the same value, a reference that has the address of a polynomial will always refer to the same polynomial, with the same coefficients, as long as the address remains constant).

Other Features:
The ability to use a wildcard capture to quickly reference the result of the last evaluation using the '_' character. '_' are automatically replaced with the automatically generated identifier of the last expression result.

The polynomial processing environment automatically names all evaluation results, even if they do not appear in an instantiation expression using the string “res0“ for the first expression, “res1“ for the second, and so on (exactly like the Scala language prompt evaluation).

The usage of the pound character ‘#’ to mark one-line comments.

The ability to specify files to be run as scripts from the interactive shell and the ability to specify to the environment that we only want to display the results of the script evaluation, instead of both the results and the script commands (though this last feature is right now only achievable by tweaking the ECHO boolean static constant of the Controller class).

Limitations:
Since our polynomial interpreter is based on regular expressions, which cannot be recursively defined, our expressions have the same character. More specifically, we cannot nest expressions within expressions. Our interpreter is thus more of a pattern matcher than a parser, really. This aspect, however does not reduce the computational power of the interpreter since we can refer to previous results using either the ‘_’ wildcard character or the automatically assigned identifiers that each expression evaluation is given by the environment (thus, for instance, instead of `(a + b) * (a + b )` as a single instruction, we can emit a sequence of two instructions `a + b` and then `_ * _` for obtaining the same result, since the underscore captures the result of the previous addition. A full grammar specification follows in the problem analysis part of the paper. This is also similar to Scala where ‘_’ is the placeholder for arguments in single input lambda expressions.

2. Problem Analysis and Model Description:

Polynomial - General Description:
Polynomials are implemented as wrappers of `ArrayList<Integer>` objects that represent their coefficients. The index of the coefficient represents its rank. Thus the integer obtained by calling the get method of the underlying array list with input k is associated with x to the power k. The evaluation of polynomials is done using Horner evaluation to achieve linear complexity in terms of polynomial degree.

Supported operations are polynomial generation using arithmetic (polynomial addition, polynomial subtraction, polynomial multiplication, polynomial division and polynomial remainders) along with polynomial generation using calculus operators (integrals and derivatives). Integrals and Derivatives are cached once generated.

Within the Polynomial class itself but not within the scripting environment we can also multiply with monomials by specifying a rank and coefficient (`monomialMultiply( 3, 2 )` would multiply by 3*x^2 for instance). We also have access to coefficients, which are returned from within the array if the access is within array bounds, or the default 0 value is returned for out of bounds access, in order to present a uniform interface to the users of the code.

We have also have access to the degree of our polynomial. the `getHead()` and `dropHead()` methods are used by the division algorithm to process the polynomial (we need to explicitly drop the head since it might be part of the remainder, and without it the recursion would not terminate). The general algorithm for division is that we calculate the coefficient resulting from the division of the highest order coefficients, and then add it to the rest of the coefficients (which are computed recursively on the current remainder of the division, which is one degree lower than the initial polynomial because of the call to the `dropHead()` method). The algorithm terminates when the current remainder has a lower degree than the divisor.

Interpreter Grammar Specification:

The building blocks of our interpreter are integers and identifiers. The definitions are as follows:

```
static final String VALID_IDENT = "[\\w&&[^\\d]][\\w&&[^\\.]]*";
```

(at least one letter optionally followed by zero or more letters or digits)

```
static final String INT = "\\-?\\d+";
```

(at least one digit optionally preceded by a minus sign)

We then proceed to define valid array literals of integer elements and valid polynomial literals of integer coefficients as follows, using certain helper definitions. Whitespace is generally ignored by appending and prepending the `"\\s*"` pattern where needed.:

```
static final String PNOM_START = "\\[\\s*";
```

('[' marks the start of our polynomial)

```
static final String PNOM_END = "\\s*\\]";
```

(']' marks the end of our polynomial)

```
static final String LIST_SEP = "[\\s*,?\\s*]";
```

(any whitespace with an optional comma marks the separation of elements in a list. this will be used by both polynomial literals and array literals)

Now we can define our polynomial literal:

```
static final String VALID_PNOM =
                   PNOM_START + INT +
                   "[" + LIST_SEP + INT + "]*"
                   + PNOM_END  ;
```

(the polynomial beginning marker, at least on integer optionally followed by zero or more integers, all separated by the list separator pattern and ending with our polynomial end marker).

We then define array literals in a similar fashion, but changing the delimitation character from '[' to a '{' :

```
static final String ARRAY_START = "\\{\\s*";
static final String ARRAY_END = "\\s*\\}";
```

(delimiters '{' and '}')

However, we also want to be able to generate, rather then simply enumerate our arrays, so we also specify a generator pattern <start> to <end> with an optional step <step_value> to guide the iteration. (for instance `1 to 10 step 2` should give us the array of all uneven digits : 1, 3, 5, 7, 9). So we also define the array generator pattern using the int pattern.

```
static final String ARRAY_GENERATOR =
                  "\\s*" + INT +
                  "\\s*to\\s*" + INT +
                  "\\s*(step\\s*" + INT + "\\s*)?";
```

(thus we can see that the step part is optional and the expression `1 to 10` yields all integers 1 through 10 - *array ranges are inclusive*).

Now we can finally fully define our valid array literal as follows:

```
static final String VALID_ARRAY =
                  "((" + ARRAY_START + INT +
                  "[" + LIST_SEP + INT + "]*"
                  + ARRAY_END +

                  ")|("

                  + ARRAY_GENERATOR + "))"  ;
```

(That is, an array is either an array generator pattern, or the array beginning marker, at least on integer optionally followed by zero or more integers, all separated by the list separator pattern and ending with our array end marker).

We then define calculus operations as any identifier followed by a `"\\.deriv"` or `"\\.integr"`

```
static final String DERIV = "\\.deriv";
static final String INTEGR = "\\.integr";


static final String VALID_CALC_OP =
            "\\s*" + VALID_IDENT + "((" + DERIV + ")|(" + INTEGR + "))\\s*";
```

(identifier followed by exactly one derivation or integration).

Thus we having defined our literal values and unary operators, we can proceed to defining instantiations as follows:

```
static final String PNOM_NSTANTIATION = VALID_IDENT + "\\s*=\\s*(("
                                     + Literal.VALID_PNOM + ")|("
                                     + Calcullus.VALID_CALC_OP + ")|("
                                     + VALID_IDENT + "))";
```

(Thus polynomials can be instantiated by either literals, calculus or by referencing some other identifier)

```
static final String ARRAY_INST = "\\s*" + VALID_IDENT
                  + "\\s*=\\s*" + Literal.VALID_ARRAY + "\\s*";
```

(Our array instances however can only take on literal values, either generated or enumerated)

Now having defined our variable instantiations we can the proceed to define polynomial evaluation and identifier evaluation. For polynomial evaluation :

```
static final String INT_ARGS = "\\s*\\(\\s*" + Literal.INT + "\\s*\\)";
```

(an integer surrounded by parenthesis)

```
static final String IDENT_ARGS =
            "\\s*\\(\\s*" + Expression.VALID_IDENT + "\\s*\\)";
```

(an indentifier surrounded by parenthesis)

```
static final String VALID_ARGS = "((" + INT_ARGS + ")|(" + IDENT_ARGS + "))";
```

(we can pass arguments to our polynomial as either integers or identifiers)

Finally an evaluation is defined as follows:

```
static final String EVALUATION =
            "(" + VALID_IDENT + VALID_ARGS + ")|(" + VALID_IDENT + ")";
```

(an identifier followed by valid arguments, or a single identifier, for cases in which we want to see the value associated with one identifier)

We must notice that type checking cannot be done at a regex level (or at least, not sanely). Type checking is done by catching ClassCastExceptions withing the code and then responding to the user with an appropriated message. That means we can try to evaluate a polynomial with another polynomial argument, but the interpreter will tell us it does not know how to do that. Arrays however *can be passed* as arguments. For array a = { k | k is Integer } the result of evaluating the polynomial P over a is P[a] = { P[k] | k from a }.

Finally we can proceed to define binary operations on polynomial identifiers or literals. We shall call there Polynomial Generators:

```
static final String PNOM_OP = "\\s*(\\+|\\-|/|\\*|%)\\s*";
```

(polynomial addition, polynomial subtraction, polynomial multiplication, polynomial division and polynomial remainders.)

```
static final String PNOM TERM =
            "((" + VALID_IDENT + ")|(" + Literal.VALID_PNOM + "))";
```

```
static final String PNOM_GENERATOR = PNOM_TERM + PNOM_OP + PNOM_TERM;
```

(Notice all are strictly binary and cannot be nested!)

3. Design Detail :

Regex Libraries:

For matching we are using the `java.util.regex` package that provides utility functions for regex string matchting. Of special interest are the `Pattern` and `Matcher` classes. We are using the `Pattern.compile( String regex )` factory method to create a Pattern representing a given regex and then the `pattern.matcher( String expression )` factory method to create matchers over specific string using the initial regex. We then use the `matcher.matches()` method to see if the pattern matches the whole expression and `matcher.find()` to see if there are substrings of the expression matched by the pattern. Calling the `matcher.group()` yields the last substring matched or found, depending on whether match or find was called. Calling group before find or match yields an `IllegalStateException`.

Organisation of Regex matching:

We use the `Expression` as a base class for all Expressions, each of which has an identifier and a value associated with it. these are accessible via the `.ident()` and `.val()` methods. Just like the Polynomial class, once instantiated, Expressions cannot have their state mutated, each Expression has exactly only identifier and one value during its existence in the program. If we really try, we could mutate the state of an underlying `ArrayList` of an array expression, but that is impossible via the interpreter shell.

We use the `Expression.construct( String expr )` factory method to instantiate Expressions. This method cascades down all the possible expression classes, trying to match them subsequently. If all matches fail, a `SyntaxError` is created, which is itself an Expression, with its value being the error message to be presented to the user.

Expression Result Storage:

Our Controller class has uses a `HashMap<String, Object>` to store the values resulting from our expression evaluations. we can use the `Controller.get( String ident )` method to access values created by our interpreting environment. To clear the namespace we simple reinstantiate a new HashMap. This can be done via the GUI as well.

For more information see attached Javadocs.

4. Implementation and Testing

Test Scripts :
Our simple polynomial interpreter has the ability to load scripts in order to ease testing and automate the generation of certain values. For example, we could use the following simple script to calculate the squares of all the integers in the interval [1; 10] :

```
# let us start by calculating all squares of ints in 1..10
# we will be using the wildcard '_' character
# it captures the result of the last expression evaluated
# this will be our data array and we shall pass it
```

```
        # to our square generating polynomial

        square = [ 0 0 1 ] # square generating polynomial
        1 to 10 # our data array
        square( _ ) # and squares pop out!
```

Notice the pound character ('#') represents comments. We can save this small script as squares.pnom and then load it into the environment and test the response :

```
        >>>  # let us start by calculating all squares of ints in 1..10
        >>>  # we will be using the wildcard '_' character
        >>>  # it captures the result of the last expression evaluated
        >>>  # this will be our data array and we shall pass it
        >>>  # to our square generating polynomial
        >>>  square = [ 0 0 1 ] # square generating polynomial

        square --> [1x^2 + 0x^1 + 0]

        >>>  1 to 10 # our data array

        res0 --> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

        >>>  square( _ ) # and squares pop out!

        res1 --> [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

        >>>
```

We can controll wether the GUI also displays the script instructions provided via the static final boolean field `Controller.ECHO`. If we need not remember the polynomials and results created by our script we can clear our namespace via the clear namespace GUI button.

We can also try the longer script:

```
        p = [1 1 1]     # start polynomial
        q = p.deriv  # get derivative of p
        z = q.integr        # get integral of q
        p - z        # test difference
        # notice only a constant factor misses due to integration
        # let us add it back
        z + _        # the underscore is a placeHolder for the difference
        z + [ 1 ]       # we can get the same result by adding a constant literal
        p - _        #notice the subtraction yields 0, p is equal to z + [1]
        # let us now test division and multiplication
        [ 0 1 ]  * [ 1 1 ]  # since our expressions are strictly binary,
        prod = _            # we must capture result via chaining
        prod / [ 0 1 ]
        prod % [ 0 1 ]     # notice result is 0, p2 divides its product with p1
```

5. Results & Conclusions

We have created a simple to use interpreter for polynomial expressions, that supports both polynomial and array literals, polynomial and array values, polynomial arithmetic operations and polynomial calculus expressions. A great factor in the simplicity of the use in the interpreter shell is due to the addition of the capture-last expression result feature via the underscore character ('_'). This allows us to rapidly chain expressions in the interpreter.

Limitations regarding nested expressions are a result of the nature of regex expressions that cannot be defined recursively. A more interesting, yet time consuming project would be to use some lexers / parsers such as a recursive descent parser such that we could have nested expression interpreted via our Shell.

6. Bibliography

- The Java Programming Language ed 4, Arnold, Gosling, Holmes *(for Regex expressions tutorial and java documentation)*
- Programming in Scala ed 2, Martin Odersky *(for describing the Scala prompt and inspiring the design of the interpreter)*
- https://docs.oracle.com/en/ *(for describing the GUI process, especially the "choose file" option used for loading scripts)*