Project 1 - Polynomial Manipulation in Haskell

- André Lima (up202008169)
- Mariana de Matos Lorenço Lobão (up202004260)

Types and internal representation

In this project, we've decided to use polynomials as a list of terms, each made up of a list of powers and a coefficient. For that, we created 3 data types and 2 type aliases:

- `Variable` is a type alias of `Char`.
- `Number` is a type alias of `Int`.
- `Power Variable Number`, where Variable is the variable at the base of a power, like "x" in "x^2", and Number is the exponent of a power, such as "2" in "x^2".
- `Term [Power] Number`, where [Power] is a list of powers that are multiplied together, such as `x^3` or `a * b^2 * c^3`, and Number is the coefficient that is also multiplied with the list of powers. For instance, the term "2 * d * g^4" has powers "d" and "g^4" and the coefficient is "2".
- `Polynomial [Term]`, where [Term] is the list of terms the polynomial is made up of. These terms are added up together. For instance, the polynomial "6 * x + 3" has the terms "6 * x" and "3".

With this structure, polynomials are both easily manipulated and parsed, since we can work with them in smaller or bigger components.

Functionalities

In this project, we allow the normalization, addition, multiplication and differentiation of polynomials. We have also implemented the parsing of polynomials.

Normalization

Assuming that a polynomial is stored in our representation of a polynomial, normalizing it consists of a few steps:

1. remove all powers with 0 as their exponent (since x^0 is 1 and 1 is the neutral element of multiplication)
2. remove all terms with 0 as their coefficient (since 0 * x^2 is 0 and 0 is the neutral element of addition)
3. multiply all powers with the same base together
4. add all terms with the same powers together

This operation is implemented in `Normalize.hs`. To normalize a polynomial, you can do the following:

```
:load Normalize.hs

-- Tests step 1.
normalize (Polynomial [ Term [ Power 'x' 0 ] 7 ])
```

```
-- Result: 7

-- Tests step 2.
normalize (Polynomial [ Term [ Power 'x' 2 ] 0, Term [ Power 'y' 2 ] 1 ])
-- Result: y^2

 -- Tests step 3.
normalize (Polynomial [ Term [ Power 'x' 2, Power 'y' 3, Power 'x' 10 ] 2,
Term [ Power 'x' 2 ] 2 ])
-- Result: 2 * x^12 * y^3 + 2 * x^2

-- Tests step 4.
normalize (Polynomial [ Term [ Power 'x' 2, Power 'y' 3, Power 'x' 10 ] 2,
Term [ Power 'x' 12, Power 'y' 3 ] 2 ])
-- Result: 4 * x^12 * y^3

-- Normalize 7 * x^2 + 0 * x^3 + 3 * x^3 + 12 * x^0 + 4 * x^2 + 7 * x^2 *
x^3 + 8 * x^2 + x^3 + 12 * x^2 * x * y^2 * x^0
normalize (Polynomial [ Term [ Power 'x' 2 ] 7, Term [ Power 'x' 3 ] 0,
Term [ Power 'x' 3 ] 3, Term [ Power 'x' 0 ] 12, Term [ Power 'x' 2 ] 4,
Term [ Power 'x' 2, Power 'x' 3 ] 7, Term [ Power 'x' 2 ] 8, Term [ Power
'x' 3 ] 1, Term [ Power 'x' 2, Power 'x' 1, Power 'y' 2, Power 'x' 0 ] 12
])
-- Result: 7 * x^5 + 12 * x^3 * y^2 + 4 * x^3 + 19 * x^2 + 12
```

Addition

Adding two polynomials can be done by simply putting a `+` sign in between them and normalizing. Therefore, our polynomial addition consists in concatenating the given polynomials with the `(++)` operator, since it's a list of terms, and then using our normalization function to normalize the result.

```
:load Add.hs

add (Polynomial [ Term [ Power 'x' 2 ] 4 ]) (Polynomial [ Term [ Power 'y'
3, Power 'x' 2 ] 3, Term [ Power 'x' 2 ] 3 ])
-- Result: 3 * x^2 * y^3 + 7 * x^2
```

Multiplication

Multiplying polynomials consists in multiplying the coefficients and concatenating the powers from each term of the first polynomial to each term of the second one.

```
:load Multiply.hs

-- Tests step 1.
multiply (Polynomial [ Term [ Power 'x' 0 ] 7 ]) (Polynomial [ Term [ Power
'x' 2 ] 5])
-- Result: 35 * x^2
```

```
-- Tests step 2.
multiply (Polynomial [ Term [ Power 'z' 2 ] 9]) (Polynomial [ Term [ Power
'y' 5, Power 'x' 6] 5, Term [Power 'x' 6] 1, Term [Power 'y' 7] 3])
-- Result: 27 * y^7 * z^2 + 9 * x^6 * z^2 + 45 * x^6 * y^5 * z^2

 -- Tests step 3.
multiply (Polynomial [ Term [ Power 'x' 2, Power 'y' 3, Power 'x' 10 ] 2,
Term [ Power 'x' 2 ] 2 ]) (Polynomial [ Term [ Power 'x' 5, Power 'y' 8]
(-7), Term [Power 'x' 2] 5, Term [ Power 'y' 3] 7])
-- Result: -14 * x^17 * y^11 + 10 * x^14 * y^3 + 14 * x^12 * y^6 - 14 * x^7
* y^8 + 10 * x^4 + 14 * x^2 * y^3
```

Differentiation

Differentiating polynomials consists in, for each term, multiplying the coefficient by the exponents of all powers in that term that have the differentiation variable as their base and then decrementing their exponent by one.

To do this, we first normalize the polynomial and then, for each term, we check if there is a power with the differentiation variable, dvar, as its base. If there is, we multiply the coefficient by that exponent and decrement the exponent by one. If there isn't, we remove the term from the output.

```
:load Derivative.hs

derivativeOf 'x' (Polynomial [ Term [ Power 'x' 2 ] 3 ])
-- Result: 6 * x

derivativeOf 'x' (Polynomial [ Term [ Power 'y' 2 ] 3 ])
-- Result: 0

derivativeOf 'x' (Polynomial [ Term [ Power 'x' 2, Power 'y' 1 ] 2, Term [
Power 'x' 7, Power 'y' 0 ] (-14), Term [ Power 'x' 1, Power 'z' 2 ] 1 ])
-- Result: z^2 - 98 * x^6 + 4 * x * y
```

Parsing

To parse polynomials given by user input we go through two steps: tokenization and parsing.

Tokenization

1. We created a new data type, `Token`, that can be:
    1. `PlusToken` corresponds to a "+" sign
    2. `MinusToken` corresponds to a "-" sign
    3. `TimesToken` corresponds to a "*" sign
    4. `PowerToken` corresponds to a "^" sign
    5. `OpenParenthesisToken` corresponds to "("
    6. `CloseParethesisToken` corresponds to ")"

7. `VariableToken Variable` corresponds to a variable, such as "x" or "y"
8. `LiteralToken Number` corresponds to a number, such as "6578"

2. The user input is read character by character. Each character is transformed into one of these possible tokens. Spaces are ignored.

Parsing

3. The list of tokens generated by lexing the user input is then parsed into a tree structure that orders operations by priority, as such:
   - `Powers`
   - `Products`
   - `Additions` or `Subtractions`
4. The tree structure returned by the parser is an expression tree, where an expression can be any of the following:
   1. `NegExpr Expr`, is translated into `-Expr`
   2. `TimesExpr Expr Expr`, is translated into `Expr * Expr`
   3. `PowerExpr Expr Number`, is translated into `Expr^Number`
   4. `VariableExpr Variable`, is translated into the variable `Variable`
   5. `LiteralExpr Number`, is translated into the number `Number`
   6. `ParenthesisedExpr Expr`, is translated into `(Expr)`
5. The parsed polynomial is simplified and normalized. All polynomials that the program outputs can be parsed again.

```
:load Parse.hs

parse "2 * x"
-- Result: 2 * x

parse "(10^4 * 7 * x + 2 * 7 * a)^3 + b^4 - 7 * x^3"
-- Result: 342999999999993 * x^3 + b^4 + 2744 * a^3 + 41160000 * a^2 * x +
205800000000 * a * x^2

parse "(x^2 + 3)^6 * (x^7 - 10)"
-- Result: x^19 + 18 * x^17 + 135 * x^15 + 540 * x^13 - 10 * x^12 + 1215 *
x^11 - 180 * x^10 + 1458 * x^9 - 1350 * x^8 + 729 * x^7 - 5400 * x^6 -
12150 * x^4 - 14580 * x^2 - 7290
```

Credits

Parser

Much of the inspiration for our parser came from the website linked below.

- "Expression Trees". *Haskell For Mac*. Accessed October 19, 2022. http://learn.hfm.io/expressions.html.