Functional and Logic Programming - Project I

Guilherme Sequeira, Pedro Ramalho

University of Porto, Faculty of Engineering, 2022/2023

Contents

1 Internal Polynomial Representation

A `Polinomial` is represented as a list of `Monomials`, which are represented by the pair `(Double, LiteralMap)`. The structure `LiteralMap` contains the literal portion of a monomial and is represented using an ordered map: `(Key -> Value <=> Char -> Nat)`, implemented using a Binary Search Tree. These data structures were chosen for the representation of polynomials since they offer the following benefits:

- Efficient search and insertion of elements
- Elements are automatically ordered, which makes it very easy to compare two `Monomials`
- `Monomials` are always in their canonical form, which facilitates the normalization of `Polynomials`

2 Functionalities

The available functionalities include the normalizition of a polynomial, adding and multiplying two polynomials together and deriving a polynomial. It is also possible to input a polynomial as a string.

2.1 Normalization

This feature transforms a `Polynomial` into its canonical form. Since every `Monomial` is already in its canonical form, the only necessary step to normalize a `Polynomial` is to sum its `Monomials` with matching literal parts (matching `LiteralMaps`).

Addition of Monomials

It is only possible to add two `Monomials` together if they have matching literal parts (matching `LiteralMaps`), otherwise the operation fails and the program terminates with an error. The resulting `Monomial`'s coefficient is the sum of the two input `Monomials`' coefficients and the literal part stays the same.

2.2 Addition

This feature sums two `Polynomials` together and returns the resulting `Polynomial` in its canonical form. Since `Polynomials` are represented internally as lists, adding two `Polynomials` together can be done by concatening both and normalizing (applying the normalization function described above) the resulting list.

2.3 Multiplication

This feature multiplies two `Polynomials` together and returns the resulting `Polynomial` in its canonical form. Since `Polynomials` are represented internally as lists, multiplying two `Polynomials` together can be done by iterating through both lists and multiplying every element from both, resulting in the cartesian product of the two lists.

Multiplication of Monomials

The coefficient of the resulting `Monomial` obtained by multiplying two `Monomials` together is the product of the two input `Monomials`' coefficients. The literal part is obtained by aggregating the literal parts from the two `Monomials`, adding the exponent of matching variables.

2.4 Derivation

The derivation of a `Polynomial` in respect to a variable is obtained by deriving each `Monomial` and normalizing the result.

2.5 Parsing

This feature takes an input string and converts it into a `Polynomial`. An error will be thrown in case of a malformatted string. Initially, a lexer function is used to convert the string of characters into a list of `Tokens` which can be one of 5 types:

1. `PlusTok`: Symbolizes the start of a `Monomial`
2. `MinusTok`: Symbolizes that the next `Monomial` is negative
3. `ExpoTok`: Symbolizes the start of the exponent of a variable
4. `IncogTok`: Symbolizes a Character. Takes a `Char` as an argument in its constructor
5. `DoubleTok` Symbolizes a Floating Point Number. Takes a `Double` as an argument in its constructor

This list of tokens is then passed on to a parsing function `tparse` which computes an `Expr`. In case of a parsing error, `Nothing` is returned instead. `tparse` takes the first token of the list and takes a different course of action depending on its type:

1. In the case of a `PlusTok`, the rest of the `Tokens` in the list are passed to the `parseCoefOrIncog` function, which is responsible for returning a `Variable` (a data structure created for this purpose only, contains a `Double` coefficient and a `Literal`.
2. In the case of a `IncogTok` or a `DoubleTok`, the first token is passed on to `parseCoefOrIncog` along with the rest of the tokens
3. Any other `Token` results in the return of `Nothing`

An `Expr` can be a single `Monomial`, represented by type `Mon`, an addition between two `Exprs` or nothing, represented by `End`. The function `parseCoefOrIncog` is responsible for returning a `Variable`, a data type that contains a coefficient and a `Literal`. It also has a negative variation which contains the same fields. In order to achieve this, the function calls several other smaller parsing functions, namely `parseDouble`, `parseChar` and `parseCharOrExpo`. The first two are self-explanatory, and the third is responsible for returning a `Literal`. A `Literal` is a data structure represented by a `CompleIncog`, composed of a `Char` and `Expo`, represented at this stage by a `Double`, a `Mul`, composed of two `Literals` or `Empty`. The function `parseCharOrExpo` is responsible for returning a `Literal`, containing all of its unknown variables (`Chars`) and their respective exponents.

Finally, some helper functions are used to convert these intermediary values to our internal representation.

1. `getLiteral`: Receives a `Literal` as an input and outputs the corresponding `LiteralMap`
2. `getMon`: Receives a `Variable` as an input and outputs the corresponding `Monomial`
3. `getPol`: Receives an `Expr` as an input and outputs the corresponding `Polynomial`

2.6 Extra

Some extra functionalities were developed for this project, most of which serve as means of achieving the project's goals. Some are listed below:

1. `toList` and `fromList`: These functions transform a `LiteralMap` to a list and vice-versa.
2. `toReadable`: Transforms a `LiteralMap`, `Monomial` or `Polinomial` into a readable formatted string.
3. `LiteralMap.insert`: Receives a `Char`, `Nat` tuple and inserts it into a `LiteralMap`: The way this is implemented guarantees that all `Monomials` are always kept in their canonical form.
4. `Nat`: Exponents are stored using this data structure, ensuring they are always positive.

3 Usage Examples

Initially, we can create two `Polynomials` by doing the following:

```
<Polinomio.hs> pol1 = parse "3*x^2*y + 4x^2*y + z*x"
<Polinomio.hs> pol2 = parse "-2x^2*y + 4*z + 3*x^3"
```

We can normalize `pol1` by typing:

```
<Polinomio.hs> poltoReadable (norm pol1)
```

which returns

```
<Polinomio.hs> 7x^2y + z*x
```

We can add `pol1` and `pol2` together by doing:

```
<Polinomio.hs> poltoReadable (poladd pol1 pol2)
```

which returns

```
<Polinomio.hs> 5x^2*y + 4*z + 1.0*z*x + 3*x^3
```

We can multiply `pol1` and `pol2` together by doing:

```
<Polinomio.hs> poltoReadable (polmul pol1 pol2)
```

which returns

```
<Polinomio.hs> -14.0*x^4*y^2 + 28.0*x^2*y*z + 21.0*x^5*y - 2.0*x^3*y*z + 4.0*x*z^2 + 3.0*x^4*z
```

We can derive `pol1` in respect to `x` by doing:

```
<Polinomio.hs> poltoReadable (polderiv 'x' pol1)
```

which returns

```
<Polinomio.hs> 14.0*x*y + 1.0*z
```

4 Bibliography and Credits

- Parsing of arithmetic expressions, http://learn.hfm.io/expressions.html
- Guidance from our professor João Fernandes
- Study materials from our professor Mário Florido