Literate Programming in Emacs and Finances

Vinícius Gajo

[2022-11-02 qua]

Contents

1 Introduction

> Literate programming is a programming paradigm introduced in 1984 by Donald Knuth in which a computer program is given an explanation of its logic in a natural language, such as English, interspersed (embedded) with snippets of macros and traditional source code, from which compilable source code can be generated. The approach is used in scientific computing and in data science routinely for reproducible research and open access purposes. Literate programming tools are used by millions of programmers today.
>
> — [1]

This repository holds an example of literate programming used in the finances landscape, implemented with GNU Emacs and some other packages (Org-mode).

The macros used in the program are written in **Python**, a very popular language, and **F#**, which is a functional-first, general purpose, strongly typed, sister language of C# and other .NET implementations.

Also, if you want to really understand and appreciate this document, make sure that you can compile some LaTeX blocks which are presented there. Those blocks were added with the goal of making it easier to understand the problem and the mathematical rationale for the function created later.

If you want to see the compiled version of this document in PDF, check this link.

1.1 Setup

When creating this example I used those software versions:

- GNU Emacs 28.2.
- The `.emacs` configuration mentioned in this repository release: link.

Disclaimer 1: Some configurations will not work, even if you have the same `.emacs` file, since it will depend on external packages (LaTeX specific packages for example), and the file system structure of your computer.

Disclaimer 2: This document is better understand if you download the `.org` file and open it in Emacs. This way you'll can see the outputs and some org configurations.

2 The Problem

In this example, we're going to tackle an investment problem, which could be stated as:

- Suppose you're going to invest your money into an application that gives you "h%" of this money invested each month, as long as you keep it there (normal investment scenario). During the time you keep it there, you can also add more money (in a month basis), and this money will follow the same rule stated before, but considering that it will start producing more only in the next month that you added it.

To make it more clear, let's use some mathematical notation.

Consider that in the first month, the money you have (z_0) is only the initial quantity you decided to invest (x_0).

$$z_0 = x_0 \tag{1}$$

Then, in the second month, the initial money will increase by a quantity given by h and you're going to increment the value adding y. So, in the second month, your money will be:

$$\begin{aligned} z_1 &= z_0 \times h + y \\ &= x_0 \times h + y \end{aligned} \tag{2}$$

In the third month you repeat the same operation. This time, your money will be:

$$\begin{aligned} z_2 &= z_1 \times h + y \\ &= (x_0 \times h + y) \times h + y \\ &= (x_0 \times h^2) + (y \times h) + y \end{aligned} \tag{3}$$

And the following months you keep doing the same operation, until the month n. By the n-month your money will be:

$$\begin{aligned} z_n &= z_{n-1} \times h + y \\ &= (x_0 \times h^n) + (y \times h^{n-1}) + (y \times h^{n-2}) + ... + (y \times h) + y \end{aligned} \tag{4}$$

The objective of this problem is, then, to discover how much money this person will have after n months in this investment adding y moneys each month.

For this specific example, I'm going to assume some values for the variables. Those are shown in the following table:

Symbol	Value	Interpretation
x_0	10.000,00	10k initial moneys
y	1.000,00	add 1k money per month
h	1,01	1% profitability
n	120	10 years = 120 months

3 The Solution

The solution for this problem is pretty straightforward since we already have derived the equations for it. Now it's just a matter of implementing it using some programming language.

Initially, my idea was to present only the F# solution in this file, but after testing some combinations of headers and other features (references [2], [3] and [6]), I noticed that the current interpreter for the F# block of code is not working properly in my system.

In the results I'm getting along the final string, the source code again and some annotations which are common when dealing with the `dotnet fsi`.

Due to it, instead of presenting only the F# block, I'm also going to present the Python block, which implements the same algorithm.

3.1 Python Solution

When using Python, it's a bit tricky to make it show the result of the print statement. In order to fix it, I found this answer in Stack Exchange pointing to the right header configuration for the block.

```
def calcMoney (x0: float, y: float, h: float, nMonth: int) -> float:
  if (nMonth == 0):
    return x0
  elif (nMonth > 0):
    previousValue = calcMoney (x0, y, h, nMonth - 1)
    return ((previousValue * h) + y)
  else:
    return 0.0

def main():
  resultMoney = calcMoney(X_0, Y, H, N)

  print(f"""
  Initial money: {format(X_0, '.2f')}
  Addition per month: {format(Y, '.2f')}
  Increase rate per month: {format(H, '.2f')}
  Investment time: {N} months
  Final value: {format(resultMoney, '.2f')} moneys
  """)

main()
```

3.2 Fsharp Solution

And this is the same algorithm implemented in F#. Notice that the result is way more polluted.

In the future I'm going to study more about this problem, and if I manage to find a solution I'll update this document.

```
let rec calcMoney (x0: float) (y: float) (h: float) (nMonth: int): float =
    match nMonth with
    | 0 -> x0
    | _ when (nMonth > 0) ->
        let previousValue = calcMoney (x0) (y) (h) (nMonth - 1)
        (previousValue * h) + y
    | _ -> 0

let main () =
    let resultMoney = calcMoney (X_0) (Y) (H) (N)

    printfn """
    Initial money: %.2f
    Addition per month: %.2f
    Increase rate per month: %.2f
    Investment time: %i months
    Final value: %.2f moneys
    """ (X_0) (Y) (H) (N) (resultMoney)

main ()
```

4 Conclusion

Literate programming is something that can have a huge impact in the organizations that adopt it, since it's easier to keep the configuration documented using it, which makes it easier to onboard other people into the system.

I got the idea to create this repository after reading the book "O homem mais rico da Babilônia" by George S. Clason. It made me start thinking more about finances, and due to it, I decided to calculate how much money I would have in some hypothetical scenarios, considering some tips from the author.

While writing this document I noticed how little I know about org-mode and babel. Then, I started looking deeper into the documentation, trying to

understand why some parts of the code did not work (F# part).

It was a very cool and challenging situation, and my goal is to keep digging into this feature in order to really understand how it works and fix my setup.

With this in mind, I plan to update this file in the future.

5 References

Finally, this last section is used to keep a record of the sources where I found most of the information required to create this project.

[1] - `https://en.wikipedia.org/wiki/Literate_programming`

[2] - `https://orgmode.org/manual/Extracting-Source-Code.html`

[3] - `https://orgmode.org/worg/org-contrib/babel/intro.html`

[4] - `https://www.offerzen.com/blog/literate-programming-empower-your-writing-with-em`

[5] - `http://howardism.org/Technical/Emacs/literate-programming-tutorial.html`

[6] - `https://orgmode.org/manual/Working-with-Source-Code.html`