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). You can find the complete documentation in reference [6].

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 understood 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:

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.

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

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 of the code execution, I'm getting along the final string, the source code again and some annotations which are common when dealing with the `dotnet fsi` (dotnet interactive environment).

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

Also, notice that the variables used in the SRC blocks were defined in this top level section, in the "#+PROPERTY:" configuration. This is a very convenient way to define common input values to the functions for different languages.

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.

```
# Python + Emacs:
# ===========================
#
# If you want to show in the end of the execution, the result of
# an operation, set :results value in the SRC block headers, and
# make sure that the main function has a return statement.
# If you want to just show a print result, set the :results output
# in the header.
#
# Reference: https://emacs.stackexchange.com/a/64539

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)

  return 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
  """

return main()
```

3.2 Fsharp Solution

And this is the same algorithm implemented in F#. Notice that the result is way more polluted (read the `.org` file to find the result).

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 ()
```

As one could see, this solution does not work very well. The result is very polluted, as mentioned before.

3.2.1 Fsharp in Shell

After testing many combinations of headers for the Fsharp SRC block, I went to the package repository in GitHub and asked the author about this behavior.

You can find more details in this link: issue 1.

In essence, he confirmed that it was happening in his setup as well, and there is not a good solution yet.

Knowing this, I started thinking in other ways to run the script. My first idea was to use the `shell SRC block` with the right shebang, pointing to the Fsharp interpreter (`fsi`).

The block:

```
# fsharp code directly:
printfn "abc"
```

Unfortunately, this configuration did not work.

One can read more about the shebang configuration in this link, from the .NET 6 release notes.

Then, I thought about a combination of steps.

First, one must tangle the Fsharp script. Then, he must use the Shell SRC block to run the `FSX` file and collect the output.

The final block is:

```
set -euo pipefail

dotnet fsi $FSHARP_FILE
```

And this configuration worked properly, although it is not very optimized (you need 2 SRC blocks in this scenario).

4 Org SRC Blocks

After some time I decided to add this section to store some information related to the Org SRC blocks. Hope this is useful for quick consults, and to get a better understanding of this cool feature.

Most of the information presented here is from reference [6] (you must read all that links to really understand Org SRC blocks).

4.1 Org SRC Operations

If you want to execute the scripts mentioned before, put the cursor inside the SRC block and hit `C-c C-c` (`org-babel-execute-src-block`). It will prompt for acceptance to run the script, so you need to type "yes".

This prompt is a security mechanism in Org, to prevent code execution. **Always check the code you're going to run in your system.**

Then, if you want to tangle the code (extract it to another file) to run in the terminal, you need to put the cursor inside the SRC block, and type `C-c C-v t` (`org-babel-tangle`).

There is also `C-c C-v f` (`org-babel-tangle-file`), if you want to tangle to a custom file.

4.2 Org SRC Headers

In order to make it easier to understand the SRC blocks headers, I decided to add this section to the document.

But notice that, according to the official documentation, although Org comes with many headers by default, there could be some added specifically to some programming language.

With no further ado, check the following table for the list of most common headers:

Header	Description	Default	Possible values
:var	Define variable values to pass the script.	-	-

Continued on next page

Continued from previous page

Header	Description	Default	Possible values
:session	The 'session' header argument is for running multiple source code blocks under one session. Org runs code blocks with the same session name in the same interpreter process. **Only languages that provide interactive evaluation can have session support.**	"none"	"none", STRING
:dir	Define the working directory of to execute the code. When 'dir' is used with 'session', Org sets the starting directory for a new session. But Org does not alter the directory of an already existing session. **Do not use 'dir' with ':exports results' or with ':exports both' to avoid Org inserting incorrect links to remote files.** That is because Org does not expand `default-directory` to avoid some underlying portability issues.	(default-directory)	STRING
:eval	The 'eval' header argument can limit evaluation of specific code blocks and 'CALL' keyword. It is useful for protection against evaluating untrusted code blocks by prompting for a confirmation.		"never", "no", "query", "never-export", "no-export", "query-export"

Continued on next page

Continued from previous page

Header	Description	Default	Possible values
:results	How Org handles results of a code block execution depends on many header arguments working together. The primary determinant, however, is the 'results' header argument. It accepts four classes of options. Each code block can take only one option per class: **Collection, Type, Format, Handling.**	"replace"	"value", "output", "table", "vector", "list", "scalar", "verbatim", "file", "code", "drawer", "html", "latex", "link", "graphics", "org", "pp", "raw", "replace", "silent", "none", "append", "preppend"
:post	The 'post' header argument is for post-processing results from block evaluation. When 'post' has any value, Org binds the results to **this** variable for easy passing to 'var' header argument specifications. That makes results available to other code blocks, or even for direct Emacs Lisp code execution.	-	-

Continued on next page

Continued from previous page

Header	Description	Default	Possible values
:file	Interpret as a filename. Save the results of execution of the code block to that file, then insert a link to it. You can control both the filename and the description associated to the link.	-	STRING
:file-ext	If 'file' header argument is missing, Org generates the base name of the output file from the name of the code block, and its extension from the 'file-ext' header argument. In that case, both the name and the extension are mandatory.	-	-
:file-desc	The 'file-desc' header argument defines the description (see Link Format) for the link. If 'file-desc' is present but has no value, the 'file' value is used as the link description. When this argument is not present, the description is omitted. If you want to provide the 'file-desc' argument but omit the description, you can provide it with an empty vector (i.e., :file-desc []).	-	STRING
:file-mode	The 'file-mode' header argument defines the file permissions. To make it executable, use ':file-mode (identity #o755)'.	-	-

Continued on next page

Continued from previous page

Header	Description	Default	Possible values
:output-dir	Combined with the 'file' mentioned before, this option is used to define the directory to store the output file.	(default-directory)	STRING
:wrap	The 'wrap' header argument unconditionally marks the results block by appending strings to '#+BEGIN_' and '#+END_'. If no string is specified, Org wraps the results in a '#+BEGIN_results' ... '#+END_results' block.	-	-
:exports	It is possible to export the code of code blocks, the results of code block evaluation, both the code and the results of code block evaluation, or none.	"code"	"code", "results", "both", "none"

Continued on next page

Continued from previous page

Header	Description	Default	Possible values
:cache	The 'cache' header argument is for caching results of evaluating code blocks. Caching results can avoid re-evaluating a code block that have not changed since the previous run. To benefit from the cache and avoid redundant evaluations, the source block must have a result already present in the buffer, and neither the header arguments—including the value of 'var' references—nor the text of the block itself has changed since the result was last computed.	"no"	"yes", "no"
:tangle	When Org tangles code blocks, it expands, merges, and transforms them. Then Org recomposes them into one or more separate files, as configured through the options. During this tangling process, Org expands variables in the source code, and resolves any noweb style references.	"no"	"yes", "no", FILE-NAME
:mkdirp	The 'mkdirp' header argument creates parent directories for tangled files if the directory does not exist. A 'yes' value enables directory creation whereas 'no' inhibits it.	"no"	"yes", "no"

Continued on next page

Continued from previous page

Header	Description	Default	Possible values
:comments	The 'comments' header argument controls inserting comments into tangled files. These are above and beyond whatever comments may already exist in the code block.	"no"	"no", "link", "yes", "org", "both", "noweb"
:padline	The 'padline' header argument controls insertion of newlines to pad source code in the tangled file.	"yes"	"yes", "no"
:shebang	The 'shebang' header argument can turn results into executable script files. By setting it to a string value—for example, ':shebang "#!/bin/bash"'—Org inserts that string as the first line of the tangled file that the code block is extracted to. Org then turns on the tangled file's executable permission.	-	STRING

Continued on next page

Continued from previous page

Header	Description	Default	Possible values
:tangle-mode	The 'tangle-mode' header argument specifies what permissions to set for tangled files by set-file-modes. For example, to make a read-only tangled file, use ':tangle-mode (identity #o444)'. To make it executable, use ':tangle-mode (identity #o755)'. It also overrides executable permission granted by 'shebang'. When multiple source code blocks tangle to a single file with different and conflicting 'tangle-mode' header arguments, Org's behavior is undefined.	-	-
:no-expand	By default Org expands code blocks during tangling. The 'no-expand' header argument turns off such expansions.	-	-
:noweb	The 'noweb' header argument controls expansion of noweb syntax references. Expansions occur when source code blocks are evaluated, tangled, or exported.	"no"	"yes", "no", "tangle", "no-export", "strip-export", "eval"

Noweb style syntax: «CODE-BLOCK-ID».

Where the CODE-BLOCK-ID refers to either the 'NAME' of a single source code block, or a collection of one or more source code blocks sharing the same 'noweb-ref' header argument.

5 Conclusion

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

In my case, I got the idea to create this project 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 a person would have in some hypothetical scenarios.

While writing this document I noticed how little I knew about org-mode and babel for SRC blocks. Then, I decided to start 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.

6 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

[7] - https://orgmode.org/worg/org-contrib/babel/languages/index.html