# LitCode

Praneet Kapoor

January 10, 2024

## 1 Introduction

### 1.1 Artistic Nature of Computer Programs

The study of computers and computations is a science. However, computer programming is an art. To understand the difference between art and science we can read the following discussion on the subject by John Stuart Mill:

> Several sciences are often necessary to form the groundwork of a single art. Such is the complication of human affairs, that to enable one thing to be done, it is often requisite to know the nature and properties of many things. ... Art in general consists of the truths of Science, arranged in the most convenient order for practice, instead of the order which is the most convenient for thought. Science and its truths so as to enable us to take groups arranges in at one view as much as possible of the general order of the universe. Art ... brings together from parts of the field of science most remote from one another, the truths relating to the production of the different and heterogeneous conditions necessary to each effect which the exigencies of practical life require[1].

A programmer writes a sequence of instructions to be given the computer so that a desired result can be achived or 'computed by the computer'. This sequence of instructions are what we call a program. He[1] has to skillfully use their knowledge about different subjects, then learn some more, and then write the program step by step. There is lots of testing and debugging involved. There is no well defined way to writing a program. Hence, it is an art.

The recent advancements in the field of machine learning have meant that large language models are able to put together a crappy code to do some stuff, it may very well be possible that we do indeed make computer programming a science. However, that would not take away the joy of writing programs. AI systems can play Chess and Go much better than most of the humans on this entire planet, but we still play those games because they are fun. Same goes for painting and photography — we can tell a system like DALL-E to cook up a painting or a drone to take the photograph of forest. However, humans will still continue to paint and click photographs just because doing so gives us pleasure.

### 1.2 Structured Programming Methodology and Literate Programming

Structured programming methodology arose as a way to introduce rigor into programming. It involves constructing a program using three basic constructs — sequential statements (which occur and are executed one after the other), a conditional statement (`if`/`else`), and iterative statments (`for` and `while` loops). Using these three basic constructs we can build simple blocks and functions which work together to produce the result we desire.

Though structured programming allows us to write programs with some discipline, the act of writing a program is still an art. Some later parts of a program might have been written before the parts which are executed at the entry point. Moving from $A$ to $B$ in a program might have involved several complex steps, which a bunch of comments won't explain well. When we see a program, we essentially see a structure which was made out of a *web* of ideas. This is the reason why we, programmers, have trouble understanding certain pieces of code which we had written a few months back. When we wrote them, we were in the *zone* and were able to craft an edifice out of scattered ideas. Once we are done with a code, we are out of that zone. It takes considerable effort to go from the edifice to the *web* of ideas which created it so that we can understand the paths to various rooms in the building.

We can remedy this by documenting our code. Every good programmer knows the utility of a good documentation. However, almost every programmer hates to write documentation for their code. Why? Why would

0.2.203 Compiled on Wed Jan 10 19:39:35 IST 2024

[1] Read 'He' as 'He/she' throughout this article

we hate to write something which will only help us and others in understanding something?

A major reason for this general dislike is the distractive nature of writing a documentation. Even with all its utility, it can become a distraction in a world where releases of a code base containing new features and bug fixes are to be shipped out at a very rapid pace.

Also, a documentation might end up *not* explaining matters clearly sometimes. After all it is explaining the structure and not the ideas which lead to the creation of the structure. This documentation will be helpful for someone who wishes to use a library in their project, but it won't be of a great help for those who wish to understand the underlying construction of the structure. (Of course, you can explain the ideas behind a piece of code in a traditional documentation, but then why not write literate programs and not break your *flow* ).

To remedy the deficiencies of 'traditional' programming and documentation, Donald Knuth came up with the concept of *Literate Programming* [2]. In literate programming, you essentially write an essay which contains 'chunks' of the source code present throughout it in an unstructured yet *natural* way. The prose explains the motivation and working of chunks. The source code inside a chunk can be expanded by referencing other chunks from inside it or by simply expanding its definition in the later part of the essay. We get the full structured program by combining the chunks together, a process which Knuth calls *tangling* as we are producing a tangled up piece out of web of ideas, something which a computer can understand easily but some person not involved in the development might have trouble understanding. To produce a documentatiom, you pass the essay through a markup processing engine like TEX or LATEX to produce a `DVI` or `PDF` file. This is what Knuth calls *weaving* the web — you weave the web to produce a beautifully typeset documentation which explains your code and its API, both for programmers and for users.

The original literate programming system was `WEB` which was used by Knuth when writing TEX typesetting engine. It used a program called `tangle` which took `.web` files and output the source code for TEXengine written in PASCAL. Another program `weave` was used to produce the documentation in `DVI` format. This documentation was later made available as a book titled *TEX: The Program*. The book is available at `CTAN`, and you can still build both TEX, the typesetting engine, and its documentation [3]. `WEB` could essentially be thought of as a bunch of macros which were built up on top of TEX. A version of `WEB` for C programs called `CWEB` was developed by Donald Knuth and Silvio Levy and still used to this day by Knuth and others [1].

Sometime later, Norman Ramsey came up with `noweb` literate programming system [4]. It was an improvement over `WEB` and `CWEB` thanks to its simplicity which arose from the simple markup it used to enclose and refer to code chunks. Code chunks were enclosed within `«Name of the chunk»=` and `@`. You could refer to another chunk from inside a chunk by enclosing the name of the reference inside double angles.
The concept of literate programming is best explained with the help of an example. Say, I want to print the first $n$ terms of the Fibonacci sequence. My language of choice is C, and here is the code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char **argv)
{
    int a, b, n;
    a = -1;
    b = 1;
    if (argc < 2) {
        fprintf(stderr, "Please provide `n' as a command line argument\n");
        exit(1);
    }
    else {
        n = atoi(argv[1]);
    }
    if (n < 0) {
        fprintf(stderr, "n >= 0\n");
        exit(1);
    }
    for (int i = 0; i < n; ++i) {
        fprintf(stdout, "%d ", a + b);
        b = b + a;
        a = b - a;
    }
```

```
    fprintf(stdout, "Done\n");
    return 0;
}
```

There is a high chance that you would have skipped the code above. I myself do that!! A literate version of the program could have been written using noweb markup-style as follows:

```
<<Fibonacci sequence generator>>=
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char **argv)
{
    <<Initialize variables>>
    <<Generate Fibonacci sequence till n terms>>
    fprintf(stdout, "Done\n");
    return 0;
}

@
```

This gives us the big picture stuff in the fewest of lines. The programmer can then proceed to explain the working of chunks, for example «Initialize variables», and the chunks which make them up. Such a documentation wouldn't *repel* attention like the plain verbose code did. Even if we were to scatter some comments through the source code, it would have still repelled us a bit.

Why?

Think about it this way: most humans tend to learn better when they converse with someone, rather than from a solo fight with the printed letter. The huge popularity of YouTube lectures can be though of as an example of that. In a similar manner, when a programmer writes a program in literate style, he tends to ruminate and explain the idea behind a piece of code. And these ideas will often come in a random order. This style of presentation, a short lecture on a piece of code, would be much more engaging than its traditional counterpart.
Of course, it is extremely important that the code is not so scattered and *littered* with prose otherwise instead of a literate program it would be a *littered* program.

### 1.3 Did Literate Programming take off?

In a way, yes. We have got Jupyter Notebooks and Mathematica, which provide *Notebook* interfaces where you write your code like you usually do but intersperse it with chunks of markdown text explaining it. It has its merits — scientific collaboration is one of them, which led to the code written in Jupyter notebooks being used to detect gravitational waves. However, notebooks aren't full literate programming systems as they do not allow you to place chunks anywhere as per your wish or refer to them.

The work most cited by fanatics in support of literate programming is *Physically Based Rendering*, a book on rendering of images in a way that imitates optics in real world, is actually a big literate program which explains the `pbrt` library. Its authors won an Academy award as their work was used in movies to render objects more realistically [5]. Another notable literate program is *Axiom*, an open-source Computer Algebra System (CAS) maintained by Tim Dally in literate programming style. He and others have also written a literate version of *Clojure*.

Except for a few notable examples, full literate programming systems took off only a bit. Which is sad knowing that had this approach to programming taken off completely, then programming would have been made more accessible to people and the quality of code would have improved drastically. Someone on ycombinator said (roughly, I don't remember the exact words), "Literate programming requires discipline. It is not suited for world where software releases are done at such a fast pace." — a statement which I found to be true once I started working in a corporate. Just think how great it would be if big corporations wrote their code in literate style — freelance programmers will be able to understand the codebase and non-technical people (like managers) could get the big picture of what a program is doing. But... transforming a huge code base into literate style is a would require too much time and effort, which I don't think any big corporation would be interested in doing.

### 1.4 Why then another literate programming tool?

I could have used `noweb` but I suffer from NIH - 'Not Invented Here' syndrome. The `-L` in `notangle` didn't seem to respect the indentation of reference in parent chunk which is not good for Python code. Its weaved output was largely unreadable for me. And the hacker guide for it was somewhat confusing — I could never figure out how I can change styles or fix the operation of `-L` flag [6].

So, I decided to write LitCode. It consists of four programs:

1. `linit`: Generates `litcode.sty` file used for styling and referencing chunks.
2. `ldump`: Prints JSON containing chunks present in files passed to it as command line arguments.
3. `ltangle`: Using the JSON generated by `ldump` it expands a chunk/reference which has been passed to it as command line argument.
4. `lweave`: Produces LaTeXequivalent of files passed to it as command line arguments. It transpiles the markup characters used by LitCode to their LaTeXcounterparts.

No one is going to probably use this tool except for me. Given my good experiences with literate programming I feel that I will be using this system for my life!! So without any more nonsense, lets write the source code for LitCode.

**Note:** Some code is present in `bootstrap/` which is used to build the current version of programs making up LitCode system. Those pieces are ran first (as you will see in the `Makefile` in the parent directory), and then the newly built LitCode is used to tangle and weave the `web` files.

## 2 Source Code

⟨*version*⟩ ≡
```
0.2.203
```

### 2.1 Terminology

### 2.2 ldump.py

Tangling a chunk involves recursive expansion of all the references that are there in the chunk. The indentation of a reference must be appended while printing all the lines of the chunk it refers to. We also need to add some markers which tell a programmer from where a piece of code lies in the WEB files. We also need to ensure that proper whitespace conversion is done — this is especially useful in the case of `Makefile` which accepts only tabs as valid indentation characters.

Most of the traditional literate programming tools tangle a chunk by going through a file, making a dictionary of chunks and then expanding the required chunk recursively. What would happen if we need to tangle 2 chunks from the same file? Some computational power will be expended in setting up the dictionary twice. To prevent this 'waste of computational power' (although in the 21st century, hardly anyone cares about it for such simple text processing job), I have written `ldump.py` . It will make a dicitionary of chunks which are present in the files which have been supplied to it as command line arguments. Then this dictionary will be printed and can be saved in a JSON file using the clobber operator, `>` . This JSON will be used by `ltangle` when tangling a piece of code.

**NOTE:** If a chunk is defined in multiple files and have no relation to each other, give them different names. This will prevent `ltangle` and `lweave` from considering them as related to each other.

We begin with the shebang line specifying the path of the python3 and import `os`, `sys`, `json` and `re` modules.
⟨*ldump.py*⟩ ≡
```
import os
import sys
import re
import json
```

A pseudocode for `ldump.py` is presented in Algorithm 1:
We will be having a function `dumper` which will take in a list of files, read them line by line and create a dictionary of chunks, `chunkDict` . This dictionary will then be converted into a pretty-printed JSON string using `dumps` method in `json` package.

**Algorithm 1:** Pseudocode for `ldump.py`

1 Check if command line args have been supplied.
  If yes: then start reading each file line by line
  If not: then throw an error
2 Go through the string and look lines satisfying the regex:
  `(?<=^<<).+?(?=>>=\s*$)`
3 Check if the chunk exists or not. If it is not their in the dictionary, then enter it. Else, we will append the lines in this chunk to the already existing lines in the dictionary.
4 Add some metadata about the position of chunk boundaries in the web files.
5 JSONify the dictionary and print it. Writing to the file will be taken care of by `>` operator.

⟨*ldump.py*⟩ +≡
```
def dumper(file_list):
    chunkDict = dict()
    for f in file_list:
        ⟨Fill chunkDict⟩
    chunkJSON = json.dumps(chunkDict, indent = 4)
    return chunkJSON
```

Our entry point into the program will be `main()` which will see if the files have been supplied as command line args or not. If they exist, we will send this list to the `dumper` . The JSON string that it gets from the `dumper` will be printed on the screen.

⟨*ldump.py*⟩ +≡
```
def main():
    if len(sys.argv) <= 1:
        print('No files provided')
        exit(1)
    file_list = sys.argv[1:]
    json_str = dumper(file_list)
    print(json_str)
```

We now have to write some code which will write `chunkDict` . When reading a file, we need to keep track of the following:

1. Line number of the current line being read.
2. Are we in a chunk or not?
   - If not, then does this line defines a chunk? Is it really new or an old one?
   - If yes, then does this line refers to another chunk? Is there a `@` in this line? (which marks the end of a chunk's definition)

We keep track of these conditions with the help of `line_count`, `inChunk`, `encounteredReference`, and `encounteredStopChar`variables.

⟨*Fill chunkDict*⟩ ≡
```
line_count = 0
inChunk = False
encounteredReference = False
encounteredStopChar = False
```

We will read all the lines of a file and store them in a list (`lines` ) in one shot using `readlines` method. `chunkLines` will store the lines of a chunk as and when we encounter. It will be off-loaded into `chunkDict` .

⟨*Fill chunkDict*⟩ +≡
```
with open(f, 'r') as fp:
    lines = fp.readlines()
chunkLines = []
```

Using a `for` loop I am now going to iterate over `lines` . The first step in each iteration would be to increment `line_count`.

⟨*Fill chunkDict*⟩ +≡
```
for line in lines:
    line_count += 1
    ⟨Fill chunkDict: Not in a chunk⟩
    ⟨Fill chunkDict: In a chunk⟩
```

There will be two conditions. Either I will be in a chunk or I won't. If I am not in a chunk, `inChunk` is set to `False` , and I have to check whether the current line is the beginning of a chunk definition or not.

⟨*Fill chunkDict: Not in a chunk*⟩ ≡

```
if not inChunk:
    chunkname = re.findall(r'(?<=^<<).+?(?=>>=\s*$)', line.strip())
    if chunkname:
        inChunk = True
```

I now need to check whether the chunk has been previously defined or not. This can be easily done with the use of `in` operator. Remember that `re.findall()` returns a list, so I need to set `chunkname` to the first element of the list that has been returned by `findall()`.

⟨*Fill chunkDict: Not in a chunk*⟩ +≡

```
        chunkname = chunkname[0]
        if chunkname not in chunkDict:
            chunkDict[chunkname] = []
            chunkLines = []
            chunkLines.append(f'@->{chunkname}: Starts at line {line_count} in {f}\n')
        else:
            chunkLines = chunkDict[chunkname]
            chunkLines.append(f'@->{chunkname}: Extended at line {line_count} in {f}\n')
```

Note how I am introducing metadata beginning with `@->` at the beginning of the chunk.

I have to now deal with the condition in which I am inside a chunk. When inside a chunk, I need to check whether the current line contains a reference or a stop-character (signaling end of the chunk), and accordingly insert metadata at chunk boundaries.

⟨*Fill chunkDict: In a chunk*⟩ ≡

```
else:
    encounteredStopChar = (line.strip() == '@')
    if encounteredStopChar:
        chunkLines.append(f'@->{chunkname}: Ends at {line_count} in {f}\n')
        chunkDict[chunkname] = chunkLines
        inChunk = False
        continue
    else:
        encounteredReference = re.findall(r'(?<=^<<).+?(?=>>\s*$)', line.strip())
        if encounteredReference:
            chunkLines.append(f'@->{chunkname}: Reference at line {line_count} in {f}\n')
            chunkLines.append(line)
            chunkLines.append(f'@->{chunkname}: Continues from line {line_count + 1} in {f}\
    n')
        else:
            chunkLines.append(line)
```

### 2.3 ltangle.py

Much of the heavy-lifting of collecting all chunks and putting them in a dictionary + putting some metadata for debugging purposes had already been done `ldump`. We now have to write `ltangle.py` which will just need to expand a chunk/reference which has been passed to it as a command line argument. Let's make a list of thing which will be passed to `ltangle` as command line arguments, along with relevant switches.

1. `-i dump/file.json`: `ltangle` will be using the dump file generated by `ldump` to read chunks and expand them recursively.
2. `-R name/of/the/chunk`: The name of the chunk we want to expand.
3. `-cc comment/line/starting/character`: Character which is to be appended before metadata lines so as to comment them out in the tangled source code. For example, `//` would be used for commenting lines in a C or C++ code. If no comment-char is provided, no metadata line will be included.
4. `-ut number/of/spaces/to/convert`: Convert specified number of spaces to tabs. Particularly useful for systems which accept only tabs (or spaces) for indentation, example: Makefile.
5. `-us number/of/tabs/to/convert`: Convert tabs to specified number of spaces.

Like before, we begin our code with a shebang. We import the same modules as before.

⟨*ltangle.py*⟩ ≡
```
import os
import sys
import re
import json

def version():
    version = '''
    <<version>>
    '''
    print('This is lweave, Version ' + version.strip('\n'))
```

I am setting `chunkDict` as a global variable. This eliminates the need to pass it as an argument during recursive expansion of the reference.
⟨*ltangle.py*⟩ +≡
```
chunkDict = dict()
```

`expandref` is the queen of this program. It takes in a single parameter — the name of the reference which is to be expanded recursively.
⟨*ltangle.py*⟩ +≡
```
def expandref(reference):
    ⟨ltangle: expandref⟩
```

Like before, our entry point will be the `main` function where we will perform much of the processing of the command-line arguments.
⟨*ltangle.py*⟩ +≡
```
def main():
    ⟨ltangle: main⟩
```

So, this would be the core structure of `ltangle`. Let's start writing ⟨*ltangle: expandref*⟩ as that is the most important part of this program. The expanded reference will be kept in `codeLines` which will be returned to `main` for further processing.
⟨*ltangle: expandref*⟩ ≡
```
global chunkDict
codeLines = []
⟨Expand the reference⟩
return codeLines
```

When expanding reference, we need to take care of the indentation of the reference. This is especially important when we are dealing with a language where indentation defines a block of code, like in Python. The pseudo-code for ⟨*Expand the reference*⟩ will be something like this:

**Algorithm 2:** Pseudocode for `ltangle.py`

1 Read a line
2 Check if it contains a reference.
**if** it contains one, call yourself (`expandref()`) again and expand upon this reference. Once you get the lines making up this reference, to each line append the whitespace with which this reference was indented. Then append it to `codeLines`.
**else** simply append the line to `codeLines`.

I will be using two variables, `encounteredReference` and `indentation` to store the current state of the function before possible recurrsive call.
⟨*Expand the reference*⟩ ≡
```
encounteredReference = False
indentation = ''
```

I pass the name of the reference to `chunkDict` and in return I get a list of lines making up that reference. I iterate over them. At each iteration I check whether the line contains a reference or not. If it doesn't, I simply append it to `chunkDict` and move ahead with the next iteration. If it does, then as per 2, I am going to call `expandref`with this reference and get its expansion. This will be stored in `buffer`.
⟨*Expand the reference*⟩ +≡
```
for line in chunkDict[reference]:
    encounteredReference = re.findall(r'(?<=^<<).+?(?=>>\s*$)', line.strip())
    if encounteredReference:
        reference = encounteredReference[0]
        encounteredReference = True
        buffer = expandref(reference)
```

Now, I extract the whitespace using which we had expanded the reference and append it to each and every line in the `buffer`.

⟨*Expand the reference*⟩ +≡

```
        indentation = line[:len(line) - len(line.lstrip())]
        for l in buffer:
            line = indentation + l
            codeLines.append(line)
```

Under the event no reference has been encountered, I simply append the line to `codeLines`and carry on with the next iteration.

⟨*Expand the reference*⟩ +≡

```
    else:
        codeLines.append(line)
```

Coming to `main`, I need to do some processing of command-line arguments to understand how the tangled code has to be formatted.

⟨*ltangle: main*⟩ ≡

```
global chunkDict
dumpfile = str()
reference = str()
commentchar = None
usetabs = False
whitespace_conv_num = 4
if '-v' in sys.argv or '--version' in sys.argv:
    version()
    exit(0)
if '-i' in sys.argv:
    i = sys.argv.index('-i')
    dumpfile = sys.argv[i + 1]
else:
    print('Cannot find -i flag')
    exit(1)
if '-R' in sys.argv:
    i = sys.argv.index('-R')
    reference = sys.argv[i + 1]
else:
    print('Cannot find -R flag')
    exit(1)
if '-cc' in sys.argv:
    i = sys.argv.index('-cc')
    commentchar = sys.argv[i + 1]
if '-ut' in sys.argv:
    usetabs = True
    i = sys.argv.index('-ut')
    whitespace_conv_num = int(sys.argv[i + 1])
if '-us' in sys.argv:
    usetabs = False
    i = sys.argv.index('-us')
    whitespace_conv_num = int(sys.argv[i + 1])
```

If all is good with the arguments that have been passed to us, we are then going to load the dump file, set `chunkDict`and then call `expandref` passing reference as an argument to it.

⟨*ltangle: main*⟩ +≡

```
# This is the portion where we can load the JSON file
chunkDict = dict()
with open(dumpfile, 'r') as fp:
    chunkDict = json.load(fp)

# Now we call the expandref function
codeLines = expandref(reference)
#print(''.join(codeLines))
```

Now, we have to format the lines — perform some white-space conversion and comment out metadata lines.

⟨*ltangle: main*⟩ +≡

```
# Now replace '@->' with comment character
lines = []
```

```
for line in codeLines:
    ⟨Perform whitespace conversion⟩
    ⟨Comment out metadata lines⟩
    lines.append(line)
codeLines = lines
```

Let us say the line was originally indented by $s$ spaces, which we wish to replace by $t$ tabs and $u$ spaces, with each tab being equivalent to $n_s$ spaces, then we have to implement the following equations:

$$t = \left\lfloor \frac{s}{n_s} \right\rfloor$$

$$u = s \bmod n_s$$

⟨*Perform whitespace conversion*⟩ ≡

```
# Spaces to Tabs
if usetabs is True:
    indentation_depth = len(line) - len(line.lstrip(' '))
    # This condition is required, otherwise for indentation_depth = 0, all
    # indentation will be eaten away.
    if indentation_depth:
        new_indentation = '\t' * (indentation_depth // whitespace_conv_num)
        new_indentation += ' ' * (indentation_depth % whitespace_conv_num)
        line = new_indentation + line.lstrip()
# Tabs to Spaces
elif usetabs is False:
    indentation_depth = len(line) - len(line.lstrip('\t'))
    if indentation_depth:
        new_indentation = ' ' * (indentation_depth * whitespace_conv_num)
        line = new_indentation + line.lstrip()
```

Commenting metadata lines is pretty simple. Just check if the line starts with `@->` character. If it does, replace it with `commentchar`if it has been supplied in the command line, else just skip it.

⟨*Comment out metadata lines*⟩ ≡

```
if line.lstrip().startswith('@->'):
    if commentchar is not None:
        line = line.replace('@->', commentchar, 1)
    else:
        continue
```

Finally, we are going to print `codeLines`. The output can be redirected to write to a file using `>` operator.

⟨*ltangle: main*⟩ +≡

```
# Now print it
print(''.join(codeLines))
```

## 2.4 lweave.py

Now we have to write the final program of the LitCode system, `lweave.py`. Like before, we begin with a shebang and the necessary imports.

⟨*lweave.py*⟩ ≡

```
import os
import sys
import re
```

Again, `main` will be our entry point in this program. `lweave` should read all the files that have been passed to it as command-line arguments, club together the lines making them, and transpile them to valid LaTeX code. This transpilation will be done by a function named `transpiler` .

⟨*lweave.py*⟩ +≡

```
def version():
    version = '''
    <<version>>
    '''
    print('This is lweave, Version ' + version.strip('\n'))

def transpiler(lines):
```

⟨*transpile from web to latex*⟩

```
def main():
    ⟨lweave: main⟩
```

First let us get ⟨*lweave: main*⟩ out of the way. If no files are provided to it, `lweave` should fail.

⟨*lweave: main*⟩ ≡

```
if '-v' in sys.argv or '--version' in sys.argv:
    version()
    exit(0)
if len(sys.argv) < 2:
    print('File names have to be provided as command line arguments')
    exit(1)
```

If files have been provided to it, lines of each file will be read and stored in a common list, which will then be sent to `transpiler` . The transpiler will return another list, containing the transpiled lines.

⟨*lweave: main*⟩ +≡

```
files = sys.argv[1:]
lines = []
transpiled_lines = []
for file in files:
    with open(file, 'r') as fp:
        lines += fp.readlines()
transpiled_lines += transpiler(lines)
```

These lines will then be joined to form a single string and will be printed on the screen.

⟨*lweave: main*⟩ +≡

```
print(''.join(transpiled_lines))
```

The transpiler needs to check whether it is reading a normal code line or a line in some listing, i.e., from `code`, `oldcode`, or `vcode` environment and take actions accordingly. The transpilations it need to perform in each of the environments are as follows:

1. **Normal mode**

   Text within `<<` and `>>` to be extracted. Check if such a reference exists. If it exists, format to `\coderef`, else continue without alteration

   `@<` and `@>` should remain unaltered

2. **Listing mode**

   Text within `<<` and `>>=` to be extracted. If `len(text)` is zero, then replace by `\begin{vcode}`. If not, then check if the reference has already been defined or not. If it is a new one, perform substitution with `\begin{code}` and `<<extracted text>>=`. Else with `\begin{oldcode}` and `<<extracted text>> +=`

   Text within `<<` and `>>` to be extracted. Substitution to be done with `@<\coderef{}@>`.

   `@<` and `@>` to be escaped properly.

Given the transpilations, here are the variables which we will use to keep track of things for us.

⟨*transpile from web to latex*⟩ ≡

```
mode = 'Normal'
chunkList = []
processed_chunkList = []
transpiled_lines = []
```

We are going to go through all the lines and make a list of chunks present in the files.

⟨*transpile from web to latex*⟩ +≡

```
for line in lines:
    chunkname = re.findall(r'(?<=^<<).+?(?=>>=\s*$)', line.strip())
    if chunkname:
        chunkname = chunkname[0]
        if len(chunkname) != 0:
            chunkList.append(chunkname)
```

We are again going to iterate over the lines, but this time actually transpile them.

⟨*transpile from web to latex*⟩ +≡
```
for line in lines:
    if mode == 'Normal':
        ⟨transpiler: Normal mode⟩

    elif mode == 'code' or mode == 'oldcode':
        ⟨transpiler: code and oldcode mode⟩

    elif mode == 'vcode':
        ⟨transpiler: vcode mode⟩

    transpiled_lines.append(line)
return transpiled_lines
```

Now let us write 'rules' of transpiling web to LaTeX in normal mode. We will first check whether we have encountered a listing environment or not and if we have, we need to set the mode and add some lines accordingly,

⟨*transpiler: Normal mode*⟩ ≡
```
chunkname = re.findall(r'(?<=^<<).+?(?=>>=\s*$)', line.strip())
if chunkname:
    chunkname = chunkname[0]
    if chunkname not in processed_chunkList:
        mode = 'code'
        processed_chunkList.append(chunkname)
        envbegin = '\\begin{code}[' + chunkname + ']\n'
        caption = '@<$\\langle$\\textrm{\\textit{\detokenize{' + chunkname + '}}}$\\rangle\\
    equiv$@>\n'
    else:
        mode = 'oldcode'
        envbegin = '\\begin{oldcode}\n'
        caption = '@<$\\langle$\\textrm{\\textit{\detokenize{' + chunkname + '}}}$\\rangle '
     + \
                    '\\thinspace +\\!\\!\\equiv$@>\n'
    transpiled_lines.append(envbegin)
    transpiled_lines.append(caption)
    continue
```

Note that I am detokenizing the name of the chunk so that special chars like _, & can be used without performing any special characted manipulations.

`code` and `oldcode` will be set if there is some text within the angled brackets. But what if there is no text? In such case, `<<>>=` will mark the beginning of `vcode` environment.

⟨*transpiler: Normal mode*⟩ +≡
```
vcodeBegins = (line.strip() == '<<>>=')
if vcodeBegins:
    mode = 'vcode'
    envbegin = '\\begin{vcode}\n'
    transpiled_lines.append(envbegin)
    continue
```

If none of these environments have started, then we are simply in *Normal* mode. Then `@<` and `@>` should remain unaltered. However, if we are encountering a reference, and if it defined in the file in the line, it should be transpiled to `\coderef` (as has been mentioned above).

⟨*transpiler: Normal mode*⟩ +≡
```
reference = re.findall(r'(?<=<<).+?(?=>>.*)', line.strip())
if reference:
    reference = reference[0]
    if reference in chunkList:
        line = line.replace('<<', '\\coderef{')
        line = line.replace('>>', '}')
```

If we are in either **code** or **oldcode** mode, we need to escape `@<` and `@>` so that they can be printed properly. Also, any references should be transpiled to `\coderef` and be escaped by the escape characters.

⟨*transpiler: code and oldcode mode*⟩ ≡
```
buffer = ''
```

```
i = 0
while i < len(line):
    if line[i] == '@':
        if i < len(line) - 1:
            if line[i + 1] == '<' or line[i + 1] == '>':
                buffer += '@<' + line[i] + line[i + 1] + '@>'
                i += 2
                continue
    buffer += line[i]
    i += 1
line = buffer

reference = re.findall(r'(?<=^<<).+?(?=>>\s*$)', line.strip())
if reference:
    reference = reference[0]
    if reference in chunkList:
        line = line.replace('<<', '@<\\coderef{')
        line = line.replace('>>', '}@>')
```

**NOTE**: The conditional statement `if reference in chunkList:` ensures that the transpilation

$$\langle\langle \texttt{reference} \rangle\rangle \rightarrow \texttt{\coderef\{reference\}}$$

is done only if `reference` is in the `reference_list`. This prevents any breaks during the building phase of LaTeX. As a consequence, of this, if you are having a chunk in another web file and do not pass it to `lweave`, then the transpilation won't happen. Therefore, either have all the relevant chunks defined in the same web file, or pass all the web files to `lweave`.

We also need to make sure that if we are encountering `@` we need to switch modes immediately.

⟨*transpiler: code and oldcode mode*⟩ +≡

```
if line.strip() == '@':
    line = '\\end{' + mode + '}\n'
    mode = 'Normal'
```

The code for transpiling `vcode` is relatively simple. Everything has to be printed out verbatim. We just need to check whether we have encountered stop-char or not.

⟨*transpiler: vcode mode*⟩ ≡

```
if line.strip() == '@':
    mode = 'Normal'
    line = '\\end{vcode}\n'
```

### 2.5 Styling file

`litcode.sty` is the styling file which will be used by LaTeXto style the chunks and references. I will be honest — I am not that good in TeXand LaTeX. Initially I took lots of help from ChatGPT to cook up this file. Later on, I decided to keep things simple, did some additions and deletions, and ended up with the current version of LitCodelitcode.sty file.

⟨*litcode.sty*⟩ ≡
⟨*Packages required by LitCode*⟩
⟨*Variables in LitCode*⟩
⟨*hyperref and url style setup*⟩
⟨*code environment*⟩
⟨*oldcode environment*⟩
⟨*vcode environment*⟩
⟨*Caption and Chunk Listing format*⟩

After lots of hits and trials I ended up with the following list of packages which will be used by LitCode. A more clever user of LaTeX can maybe add and delete a few packages from this chunk.

⟨*Packages required by LitCode*⟩ ≡

```
\RequirePackage[T1]{fontenc}
\usepackage{amsmath}
\RequirePackage{textcomp}
%\RequirePackage{inconsolata}
\RequirePackage{calc}
\RequirePackage{caption}
```

```
\RequirePackage{hyperref}
\RequirePackage{cleveref}

\RequirePackage{fancyvrb}
\RequirePackage{listings}
\RequirePackage{xcolor}

\RequirePackage{xparse}
\RequirePackage{subcaption}
```

Let me first tell you about the environments. There are three flavours of environments in LitCode, all of them being derived from `lstlisting` environment:

1. `code`: Used when defining a new chunk. Caption and labels will be set for this environment. Captions will be invisble so that no weird indentations or spaces are there between normal text and listing. `@<` and `@>` are used to escape inside this environment so that we can refer to other chunks.
2. `oldcode`: Same as `code` but used when extending the definition of a previously defined chunk. No caption and label is defined for this environment.
3. `vcode`: Used for writing a purely verbatim piece of code. No captions, no labels, no escape characters.

For these environments, I have defined a few variables which will allow us to set listing font, background color, margin and textwidth, although the latter two aren't used at all (they are just relics of a once complicated `.sty` file that I had).

⟨ *Variables in LitCode* ⟩ ≡
```
\def\litcode{LitCode}
\def\linewidth{\textwidth}

\def\xleftmargin{10pt}
\def\xrightmargin{0pt}

\def\litcodefont{\fontfamily{SourceCodePro-TLF}\small}
\def\codebg{yellow!15}
\def\vcodebg{green!20}
```

I also wanted to enable 'jumps' whenever I would click on reference. I also wanted these references to be colored differently. Thats why I have set `colorlinks` as `true` and have set both `linkcolor` and `\urlcolor` to `red`.

⟨ *hyperref and url style setup* ⟩ ≡
```
\hypersetup{
    colorlinks=true,
    linkcolor=red,
    urlcolor=red,
    citecolor=red
}
```

Thanks to the line below, if you are using a URL, it will be put in the same style as the surrounding text.

⟨ *hyperref and url style setup* ⟩ +≡
```
\urlstyle{same}
```

Now come the interesting chunks, which troubled me a lot — definition of custom environments, caption and chunk listing style. I will begin with the queen — the `code` environment.

⟨ *code environment* ⟩ ≡
```
% Define code environment: Used when defining a chunk for the first time
\lstnewenvironment{code}[1][]{
    \vspace{-1em}
    \lstset{
```

⟨ *code environment* ⟩ takes in one argument which is used to set both the caption and label of the environment. Note how I am using `\protect\detokenize{}` to make sure the formatting is not messed up. Do note that captions will be invisible so that we can have captions and code chunk indented by the same level. Also 'fake captions' will be inserted to by `lweave` with proper formatting so that chunk definitions and extensions can be distinguished from each other.

⟨ *code environment* ⟩ +≡
```
        caption={\protect\detokenize{#1}},
        label={#1},
```

We further extend ⟨*code environment*⟩ by setting `columns` to `fullflexible` so that the letters are nicely typeset (unlike in `fixed` column width). I have set the font to `\ttfamily` — you can change it to something else. `keepspaces`, and `breaklines` are used to make sure the indentation is not changed and the lines are wrapped if they go over margins, respectively. `upquote = true` ensure that double quotes are formatted as simple upquotes only (and not as “ and ”). `background` is used to set the background color via `\codebg` variable: set it to any color you wish in the preamble.

⟨*code environment*⟩ +≡

```
        columns=fullflexible,
        basicstyle=\litcodefont,
        keepspaces=true,
        breaklines=true,
        upquote=true,
        backgroundcolor=\color{\codebg},
```

The interesting part is the `escapeinside` variable, which is used to set the characters which will be used to escape inside the `lstlisting` environment. I have set these variables to `@<` and `@>`. To print these variables inside the environment, they need to be escaped themselves. Example:

```
@< --> gets transpiled to --> @<@<@>
@> --> gets transpiled to --> @<@>@>
```

⟨*code environment*⟩ +≡

```
        escapeinside={@<}{@>}
    }
}{}
```

We can define `oldcode` and `vcode` environments in a similar fashion with a few minor changes: in `oldcode` we will get rid of caption and label, and in `vcode` we will get rid of escape characters.

⟨*oldcode environment*⟩ ≡

```
% Define oldcode environment: To be used when extending an already defined chunk
\lstnewenvironment{oldcode}[1][]{
    \vspace{-0.5em}
    \lstset{
        columns=fullflexible,
        basicstyle=\litcodefont,
        keepspaces=true,
        breaklines=true,
        upquote=true,
        backgroundcolor=\color{\codebg},
        escapeinside={@<}{@>}
    }
}{}
```

⟨*vcode environment*⟩ ≡

```
% Verbatim code environment: Takes no captions, has no labels and won't get included in the
    listing.
% Purely verbatim!!
\lstnewenvironment{vcode}[1][]{
    \vspace{-0.5em}
    \lstset{
        columns=fullflexible,
        basicstyle=\litcodefont,
        keepspaces=true,
        breaklines=true,
        upquote=true,
        backgroundcolor=\color{\vcodebg},
    }
}{}
```

We end this `.sty` file by defining the format in which we want to display captions in references and in listings. I have followed the style which had been used by Knuth and Ramsey:

- ⟨ *name of the chunk* ⟩ ≡: Used when defining a chunk for the first time
- ⟨ *name of the chunk* ⟩ +≡: Used when extending the definition of a chunk
- ⟨ *name of the chunk* ⟩: Used when referencing a chunk

The first and third type of formatting is defined in `litcode.sty` itself. One can cook up a LaTeXcode which checks whether a chunk has been previously defined or not and accordingly format its captioning. I am not that clever. This decision (checking the *freshness* of the chunk and formatting its caption) is done by *lweave*.

⟨*Caption and Chunk Listing format*⟩ ≡

```
\DeclareCaptionFormat{codecaptionformat}
{
    \phantom{$\langle\textit{\detokenize{#3}}\rangle\!\!\equiv$}
}

\captionsetup[lstlisting]{
    format=codecaptionformat,
    justification=raggedright,
    singlelinecheck=off,
    aboveskip=-2em,
    belowskip=1em
}

\makeatletter
\renewcommand\l@lstlisting[2]{\@dottedtocline{1}{0em}{2.3em}{\textit{#1}}{#2}}
\makeatother

\newcommand{\coderef}[1]{$\langle$\textrm{\textit{\nameref{\detokenize{#1}}}}$\,\rangle$}
```

Note how in `\DeclareCaptionFormat` I have set the caption under `\phantom` command so that they are invisble and do not mess up the indentation. In `\captionsetup` I have setup `skip` to `-\baselineskip` so that the vertical space between a chunk and lines of normal text preceding it is not huge. Also look at the use of `\detokenize`: it is being used to make sure that special chars do not mess up the formatting of captions and references.

## 2.6 plsty.py

`plsty.py` or simply `plsty`, will be responsible for writing `litcode.sty` to a directory which is a passed to it as a command line argument. Of course, I can upload this styling file CTAN, but I do not think that this is anything ground-breaking, so I will keep it with myself.

⟨*plsty.py*⟩ ≡

```
#!/usr/bin/python3
import os
import sys
import time
import subprocess as sbpr
```

The entire content of `litcode.sty` will be present inside `plsty.py` as a multi-line raw string. Why a raw string? This is because this `.sty` file makes extensive use of backslash \. I do not want to make things messy by escaping every backslash... Better to just use a raw string. :-)

⟨*plsty.py*⟩ +≡

```
litcode_sty_content = r'''
⟨litcode.sty⟩
'''
```

And that is something I might not have been able to do without using literate programming: reuse the code for a file as a chunk inside another source file.

`plsty.py` will accept only a single command line argument which would be the name of the directory where `litcode.sty` is to be saved. The 'entry-point' for our script will be the `main` function, where the command line arguments will be processed to get this directory name. If none is supplied, then the program should exit with an error code.

⟨*plsty.py*⟩ +≡

```
def main():
    global litcode_sty_content
    if len(sys.argv) <= 1:
        print('Please supply a directory name')
        exit(1)
    directory = sys.argv[1]
```

We can also ensure that the directory is made using `mkdir -p`. It won't do any harm to the directory's contents if it already exists.

⟨*plsty.py*⟩ +≡

```
    command = f'mkdir -p {directory}'
    os.system(command)
```

Finally, we write the raw string to `litcode.sty` placed under the directory supplied as a command line argument. We need the full path to `litcode.sty` which can easily be made by appending the name of the directory (relative to the current directory/position in the terminal), system path separator (which we get by `os.sep()`) and the string `'litcode.sty'`. I have just used a formatted string, although these strings could also have been appended by `+` operator.

⟨*plsty.py*⟩ +≡

```
    full_path_to_sty = f'{directory}{os.sep}litcode.sty'
    with open(full_path_to_sty, 'w') as fp:
        fp.writelines(litcode_sty_content)
```

## Listings

## References


[1] D. E. Knuth, "Literate programming, volume 27 of," *CSLI Lecture Notes*, 1992.

[2] D. E. Knuth, "Literate programming," *The computer journal*, vol. 27, no. 2, pp. 97–111, 1984.

[3] D. E. Knuth, "Tex: The program," *Reading: Addison-Wesley*, 1984.

[4] N. Ramsey, "Literate programming simplified," *IEEE software*, vol. 11, no. 5, pp. 97–105, 1994.

[5] M. Pharr, W. Jakob, and G. Humphreys, *Physically based rendering: From theory to implementation*. MIT Press, 2023.

[6] N. Ramsey, "The noweb hacker's guide," *Princeton University*, vol. 9, 1992.