Literate Programming Project

Pitambar Sai Goyal <lasawfwiah@gmail.com> Mon, Dec 16, 2019 at 3:12 PM
To: smithstrivedi.21IT@licet.ac.in

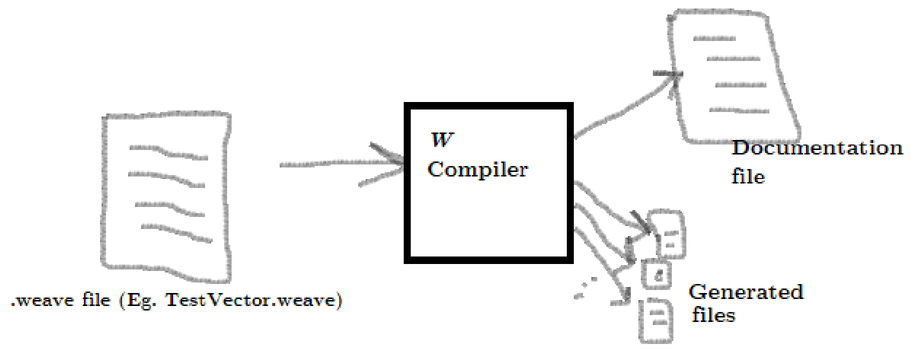
Hey.
Two things:

I) **Software description**

1. Basics
i) We are to, essentially, *implement a programming language*. This perspective helps us understand *what* the software project is- we have to write a program/compiler that will read instructions written in a file, process them and execute them. These "instructions" will be described below. The chief drawback of this perspective lies in the fact that our "programming" language cannot be used like other programming languages, but has limited scope, rather like HTML and SQL. The reason why we need such a language instead of just using ordinary programming languages is the same reason we have HTML, SQL etc.: for this particular *application*, it would be very inconvenient and counter-intuitive to use the usual programming languages to do a simple task needed by this application.

ii) This software is in the same software category of IDEs like NetBeans, Eclipse, Jupyter, Rational Rose etc.,- a tool to help programmers and software engineers in their work. The idea in this case is to allow the developers to do everything without needing too many tools and interfaces. The way code is written today, we get a lot of code which *cannot be read, maintained or repaired*, because even the programmer cannot remember why or how he wrote certain segments of code. A programmer can use this software as though he/she is writing an article and he/she can write the operations needed to do each task/sub-task in the middle of the article just like one writes formulas and operations in an article. The compiler reads all these operations mentioned in the article and combines them to form actual executable computer programs, as or when the programmer desires. The programmer can also combine these operations as and how he/she pleases, to perform various bigger operations and reuse them. The result is summarised by Knuth-
"*The practitioner of literate programming can be regarded as an essayist, whose main concern is with exposition and excellence of style. Such an author, with thesaurus in hand, chooses the names of variables carefully and explains what each variable means. He or she strives for a program that is comprehensible because its concepts have been introduced in an order that is best for human understanding, using a mixture of formal and informal methods that reinforce each other.*"
The result is that ideas which have been used for hundreds of years in literature (especially scientific literature) can be reused in this,- one can have an index and contents in order to help the reader find various code segments.

3) The program/compiler does the following. It reads a "program" or file which the programmer has written, containing his thoughts and also his operations (code), and generates- i) A *documentation* file, (like all the .pdf files in the previous mail I sent you), ii) A set of one or more *output files*- which have been created as the output of the program. Usually, these will be program in some old language (like the C/C++ files in my previous mail.) If you take one of the examples in the previous mail, you will understand the mechanism. (The overall mechanism of the new software will be almost the same, but very different in use and in implementation.) I don't know if you checked out any of the files ending with ".weave" in my previous mail- you have to download them and view them with Notepad. Then you will see programs written in my older, basically *NOT GOOD* language. The compiler processes this and generates the files as described above-



The documentation file is also hard to generate- we have to print both the operations (code) and the text similar to the kind of printing seen in scientific literature (see TestVector.pdf as example.)

2. Details

Now, you notice that I have indicated that the generated files need not contain code. What is going to be done is the following.
A file is simple a string of bytes stored in the hard drive. Like strings in RAM, they have names, a beginning byte and an indicator of the end (i.e. "**eof**".) But since the hard drive is built in order to carry larger amounts of data, files are stored somewhat differently than strings of bytes in RAM. While writing programs involving strings, we have to be careful about not wasting or misusing the RAM,- we have to beware of fragmentation. But with files, we do not have this worry because operating systems take care of this very comfortably (remember the "linked-list" storage scheme we studied in OS?) So we can handle files like we handle strings, but much more freely in some aspects.
In our language, we can describe files like strings very comfortably,- initialise them, concatenate them, modify them etc. This has the side effect that we can treat even program code like data and manipulate it, if needed. (There are so many programming paradigms made to do this,- functional programming, currying, LISP etc.)
For this reason, the most important instructions are as follows:
1) *Initialise file*. Since we can more freely name files, unlike strings in C?C++/Java etc., we have to initialise them differently. For various reasons, I have made the initialisation instruction like:
@<....*File name*.....>= ....*contents*....@/
and this will show up in the documentation in various ways. The "@" comes at the beginning of all instructions, which is not only the convention used by many LP softwares so far, but allows us to us the notations after the "@" freely elsewhere, without having to use some escape sequence. Meaning, if a programmer is using the software, he will probably be using the "less" than or "greater-than-or-equal-to" symbols ("<" , ">=") in his program a lot. So we need to use this "@" to specify what is instruction and what is not. The "@/" symbol specifies the end of the file contents.
2) *Add to end of file*. This "concatenates" an existing file with new material at the end of it. The instruction is as
@<....*File name*....>+= ....*contents*....@/
3) *General concatenation*. This is different from the above instruction, because we need to be able to use other files specified elsewhere and join them in any way we please. The way we will do this is by simply writing the file name within a type of brackets in the middle of a string, like:
@<....*File name*....>=....*contents*....@<....*Another file*....>....*contents*....@<....*Another file*....>....etc.
4) *Macros with arguments*. This can be understood if you understand C/C++ macro functions. These are extremely powerful tools in programming, and any other engineering really, and do not get enough attention. They are strings that are not fully specified, with some dummy variables in between than can be specified later. For example, in C/C++, we can define:
#define f(a,b) a+b
and we also do not have to specify what type *a* and *b* belong to. We can, in the program, write-
c= f(s1,s2);
where c, s1, s2 can be of types float, int or even string (in C++). The compiler simply copies and pastes whatever is given in the macro **#define** definition, and changes the a and b into s1 and s2. In fact, I used to write many C/C++ programs without any "**int main(){**" or closing bracket "**}**", and simply import another file where I had defined many macros, including:
#define start int main(){
#define stop }
and write:
start
...program code....
stop

In our software, the macros will be specified like this.
@<....*File/code segment title with variables in between* |...|.....>=....*contents or code segment with variables in between* |...|....@/
For example:
@< Add |a| and |b| >= |a| + |b|
Compare with the C/C++ conventions above.
5) *General string functions*. There must be a set of instructions implementing the various string functions in the various string libraries we have, and apply them to files, as above. I have not thought of any strict convention in this regard, but I think only a few more of these functions may be necessary,- such as inserting a string at a certain point etc.

II. **What to start with**
The reason this software is difficult to implement is that it has to be implemented at the low-level,- i.e. using C/C++ and no object orientation. That is the only way it will succeed without crashes.
I think you have to follow the steps of the prescribed software process as specified in your OOAD/Software Engineering courses, and that is not such a bad idea. You may be asked to begin writing down the use-case models and preliminary design using *conceptual* classes etc. This is a good way to begin,- but the answer to the problem of beginning planning is already half-given to us: the steps in compiler design. It is NOT a good idea to simply follow the steps of compiler design as prescribed in textbooks, because that will be completely wrong for any given application. It would be an even bigger mistake to try to use any of the tools available,- like *Lex* and *Yacc* for implementation.
Instead, the compiler design textbooks give a good way to plan out our design. We can:
1) See their prescriptions (especially the diagrammatic representations of the flow)
2) Adapt to our purposes (i.e. our application)
3) Cancel out unnecessary steps

Your staff will recommend the so-called "Dragon-book" by Aho, Hopcroft and Sethi for checking out these prescriptions, but I would recommend "Engineering a Compiler", by Torczon and Cooper.

If you are completely unfamiliar with the underlying notions of the subject, you should try beginning with the following,- it will also be helpful to the project: Try writing down how (or try writing down the steps by which) a program may read an assignment statement of the above type and actually create a file with the contents specified in the statement without any errors. (There can be many errors, and we may not think of all of them in the beginning,- for example, the file name may contain characters not allowed in file names, like ?, | etc.)

And I think that's it. Reach me on WhatsApp for any clarifications.