Compilers Practices Requirements Document

Compilers are complex programs because they use highly technical data structures and theory, and require highly recursive thinking. So the program structure and design is key to build them. In this course, we just develop small portions of a compiler. But we put emphasis on how this code is designed.

In this compilers course, we are interested in building programs that are generic engines. This means that they work with any given input and provide an output depending on the features supported by the program. How many features are supported can evolve over time, and beyond the project assignment if we just do one part of the whole compiler layer. But the code has to be an extendable engine to support the missing features. It cannot be a hard-wire specific solution for a particular input example, that to have a whole set of features it needs to be started again from scratch. A generic engine means that the program has to be parametrized and must interpret the input as a generic syntax sentence and not as a literal fixed string of words. This is explained in the handouts and you'll be guided in the team meetings to apply this strategy. It is very important to follow the forbidden code requirements listed below.

The practices are highly conceptual, so you have to think and design before doing code. The team size is large (6 people) to impose you to split the overall functionality in smaller pieces as each team member must be responsible of a part of the project. The objective is to learn to work as a team efficiently as this is also an integral part of the task of programming. Identify independent parts of a project so that each person can work independently without overlaps and conflicts is a programming design task. So work assignment to team members is a programming task too. Good communication of each project contributor to the rest of the team is a programming task too. Each member must update the rest of the team of conceptual needs or definitions to discuss with the team how to handle this new conceptual need inside the overall project design and code. It is not a good programming practice to just keep doing code by your own until you are done with what you are assigned to do. Instead, take your module's responsibility as a thinking task to design what is needed to obtain what it must do. Think initially that someone else in the team will do what you propose or design (although this someone else may end up being yourself). So your starting task is to define data structures and function definitions (just the specification not the implementation) to discuss with the team. The team have to decide how all these functions relate and integrate in the overall design and then once decided, each member gets an independent portion of this overall design to develop. So always there has to be a team view, more than individual views, of the project. This is done continuously and revised if needed in each team meeting. Every time that something new is identified it needs to be proposed to the overall design to decide how to handle it in the global view of the project. As the project advances, less undefined things appear and the contributor's tasks gets more coding intensive than design intensive. An efficient team advances fast in this iterative process and converges easily to a compact code with no overlaps nor conflicts and hence assigning clear tasks that integrate with the overall project and can be worked out independently by any of the team members. This communication is what makes build the overall understanding of the solution, avoid overlaps and integration problems, and tune a good integrated project design. Note that each project has a different design and there are no general rules to identify a project design, so we have to learn by experience.

Without experience, the coding tasks are the means to identify design issues as they cannot be identified in advance. But, even then, do not assume it is too late to bring it back to the team and include it to the overall project perspective. If you do you build experience in identifying the design things, so next time you'll be able to identify them before getting deep in to the implementation.

The complexity of these practices imposes you to think the design. So they are a good framework to practice the team programming abilities just described. So the course objectives include these abilities. To guide these abilities, we impose the following practice requirements that must be followed at any time, from the first reading of the practice assignment, through the design process and development of intermediate code versions, to the final submissions and in all documentation and team meetings materials. The objective is that all team member acquires these programming team habits so they become your unconscious way of working. You will notice the efficiency effects as a team and the design quality of your outcomes as you progress. It will be difficult to do in the first practice, but it will be much better to apply in the second practice, and we expect a significant improvement in the third practice. Although the work efficiency cannot be evaluated, it is implicitly evaluated in the time you take to resolve the practice assignment and in the quality of their internal designs.

So it is very important that you don't take these requirements as imposition of the submissions and just polish the code and documents before the submission to comply with the requirements. If you do it this way, it loses its purpose as it becomes extra work instead of time efficiency. The assignments will be much more complex to do and a team of 6 is too many people to work with.

With the programming experience we have, we impose the requirements listed in the table below to the practice projects. This list is a compendium of good programming practices applied to our context. You have to strictly follow all of them, but if you want to do something differently just comment it in the official team meetings and if you get the faculty approval you can use it

as you describe and document in the project material. The approval is not enough, any deviation agreement of these requirements has to be clearly stated in the documentation so that the entire team and external people have it documented and clear.

Criterion	Definition	Exceptions
Code Requirements		
1. Structured global variables	Global variables have to be the core of the data structure of the program. So they have to be minimized, organized and structured (structs). Extra variables that are just for temporary/local use should not be global and are forbidden. So except for a small set of very well structured global variables agreed initially by the team and in the first review meeting, no other global variables are allowed. Define the program core data structures early on in the design (with the empty project template indicated below), so all team members work with them and all with the same.	Agreed in review meetings
2. No magic numbers	Numerical values are forbidden. Any number needed in the code should be defined as a constant identifier named in relation of the usage of the number. The code cannot have any number except in the #defines. All have to be constants or variables.	0 and 1 have some exceptional cases, like incrementing a counter (equivalent to ++) or initializing a variable.
3. No magic chars nor strings	Char and string values are forbidden. This is very important to get your program to be a generic engine of a portion of a compiler. The code cannot have constant strings or chars (the same as the magic numbers), except in the #define. All have to be constants or variables.	None
4. Language specification files	The solution must be able to change the (input) language specification and all language dependent constants and specification must be concentrated in a file, either a .h file (language-spec.h) or input file (language-spec.txt) or a mixer of both. But the rest of the module files should not have language-depend information.	None
5. Small functions	Functions longer than one display are forbidden. Functions have to be small and focus and have to implement one (single) clear responsibility/functionality. For more than one functionality create a wrapper function to call two (or more) other functions. All functionalities must be defined as a function (stop the long endless sequence of instructions, give structure to your code with functions).	None
6. Code duplication	Code duplication is forbidden. It is forbidden to have similar functions doing the same (or similar thing) or having similar code several times in different parts of the program. You must identify what functions to create and call them all places that need to do this functionality Code duplication can be avoided by defining the correct function with appropriate parameters to call it in the different places where the same thing needs to be done with different options or values of the parameters. Communicate this in advance to your team mates so they use the same function you define or they provide you the function already defined. Copy-pasting of code is strongly discouraged because it leads to code duplication. Only use it, to move a particular code out of a function to create a new function. So every time you want to do copy-paste is an indication you should consider doing a function.	None.
7. The function main()	The main function of your program has to be a wrapper function to call other functions. There should not be direct code but just function calls. This way the main is almost a pseudo-code of the program structure. A main function with a long list of (direct, not function calls) instructions is forbidden.	None

Criterion	Definition	Exceptions
8. Define modules	Your code must have at least as many modules as team members. Each team member must be responsible to design, implement and test at least one module. Group the set of functions related to similar functionalities (typically manipulating a data structure) in a module with separate code files (.c, .h) to be worked independently. The modules should not correspond to the list of features requested in the handout, but instead they have been related on how the program internally has to be organized. It is forbidden to have the module names to be the handout functionalities. The internal structure of a program is not driven by how this program is presented or used.	Minimum n modules for a team of n members
9. Project template	It is compulsory to have a running project at all times. Start with a project template with all initial functions defined as empty functions so the project compiles and runs (printing informative messages) and divide the functions in modules so each team member is owner of files so free to modify without conflicts and advance the project independently. This project template must have the data structures of the program as all modules must use them. In team meetings, we will review the current status of the project and the committed version in the github server has to compile and run always. Each person can have his/her working version in any status.	None
10. ON/OFF Test traces	Each functionality must use an ON/OFF trace system to verify each part independently. So each part has to have a different flag to turn on or off. See project template as example.	
Documentation Requirements		
11. README	Maintain and update a useful README project. You decide what to include in it. But it must be informative to the team working on it, and also to the external people to the project (listener's team and faculty). It must include in the way you prefer at the least the following content: the project design principles and conceptual structure, the data structure explanation and justification, the project structure (modules, and leader names), explanation of how the modules interact between them, and explanation of the internal design of each module, work assignment history and the list of team leaders for all official team meetings in advance.	
12. File documentation	A comment header is required at the top of every source code file (e.g., .c, .h). The comment must contain: program name, author(s), creation date, description of the functionality, and any required information needed to use and understand the content of the file.	None
13. Data Structure documentation	A useful comment is required for each declaration of a data structure (struct) and each and every constant and field of an struct included in the declarations.	None
14. Constants documentation	Each constant (#define) must have a meaningful name and a comment explaining the meaning and/or use of this constant. Remember that we may have several constants with the same value (zero for example) but the constant has a different interpretation and context use.	None
15. Variables documentation	ALL important variables must have a meaning and comment the same as the constants. The exception are only temporary variables that are just manipulation variables. Still try to give them a meaningful name but you can omit a comment next to it.	Temporary variables in a function (variables with scope just the visual display)
16. Function documentation	A useful comment is required for each function, describing what it does, its parameters, return value, and when needed, how the function works.	None
Team meetings: 1h meetings with two project teams (20-25 min each project)		
17. Presenter Team	The team must present its work explaining the conceptual design first, and go to explaining each of the parts/modules next. It is the responsibility of the team to make it understand to the other team and faculty the intention of the work. You have to prepare the material you need in the meeting and submit it in the meeting pre-submission for P1 to P6 meeting. The meetings start at conceptual level and depend in the level of detail if necessary or if there is time. Each member of the team must talk in the meeting to provide an update of his/her responsibilities. The presenter team swaps to listener team role when the other team presents its work.	

Criterion	Definition	Exceptions
18. Listener team	You are responsible to conceptually understand the presented team work and provide feedback of what you understand and makes conceptual sense to you, what you don't understand or don't make sense to you and inform them if they comply with this requirements document. You are supposed to ask questions if you need any clarification. You can think as having the role of project's client. Each member of the listener team has to fill a feedback sheet with his/her personal view of the presented project. This form is delivered to the presented team and faculty as feedback record of the session/meeting.	
19. Project's (meeting) leader	The project leader leads the meeting presentation and presents the design of the project as a whole (of the presented project). The project leader must be different at each team meeting, rotating the role. All team members must be project leader at least in one official team meeting. Along the course there are 6 compulsory/official sessions, so each team member can be leader once. If there are any extra consulting meetings there has to be a team leader too to lead the meeting.	
20. Module's leader	The person in charge of the designing and developing a module is the module's leader This person must explain conceptually the module in the team meetings explaining how this module relates to the entire project, what functionality this module defines, how is designed, implemented and tested, and what is the current status of the module. This corresponds to the team members work responsibility inside the project. Therefore, this role remains fixed for the entire project unless the team decide to divide tasks differently. If the team defines more modules than members, a member must curry more than one module's responsibility. However, there has to be at least one module per member to be accountable for it and have individual responsibilities to perform.	
21. Conceptual Design slides	In addition to the normal documentation described above, you are required to have a slides presentation with a graphical representation of the conceptual design and components.	
Team work Requirements		
22. Team work assignment	Assigning what to do to each team member is a programming act as it decides what each team member must do and in a sense what modules the program will have. So, it defines how the program is going to be structured. If you assign work randomly you are defining the program structure randomly. If you assign work based on the handout specification, you define a program structure based on the use of the program. But the internal structure of the program is not necessarily driven by this, but instead based on how to store and manipulate the data. The program structure is data driven. So, assign initially the work to team members as you can. But this assignment has to last just to do the first thinking to decide the design and structure of the whole work, and then reassign tasks based on the needed project structure. Each member work assignment has to be clearly stated inside the code (header of files), in the design slides and the README must have a section on it too and any other place you feel useful. If an assignment is changed, keep the history of responsibilities (not just overwrite the assignment).	
23. Github project	It is compulsory to use github to work the code as a team. Invite dolors.sala@upf.edu to your project. Use the project template as starting point. Create at least as many modules as team members, or more as you need. You can decide to have a different project for each practice, or if you plan to make the 3 practices to work concatenated, to have a different branch for each practice inside the same project. You can decide at each practice assignment.	
Compilers Theory Requirements		
24. Recursion	Have in mind that language specifications, and hence compilers to interpret the languages, are highly recursive. As example, the body instructions of an if is specified exactly the same than the body instructions of a for or a while. Or a condition of an if is specified exactly as the for condition and while condition. So we should not work out independently each instruction with no interaction between them. Instead we should analyze the instructions we must support and identify what they have in common and what different parts of instructions are derived from the whole set of instructions to be supported/implemented.	
25. P1 Preprocessor	You can implement this practice with no theory requirements of compilers course. See handout for full specification and additional particular requirements.	
26. P2 Lexer	The core engine of the lexer must implement an automaton (matrix and so on, see theory and seminar solution). See handout for full specification and additional particular requirements.	
27. P3 Parser	The core engine of the parser must implement a shift-reduce automaton. See handout for full specification and additional particular requirements.	
Additional Requirements		
28. Use of AI	You can use AI engines to help you design and code. However, you have to know perfectly your program with not a single hesitation. If you submit or have a piece of code in a team's meeting that you don't know exactly how it works, it will be considered plagiarism and the entire project and team will be accounted for it. So be careful when using AI. It can assist you and teach you, but you still are the center of the work and hence it cannot do the work for you.	