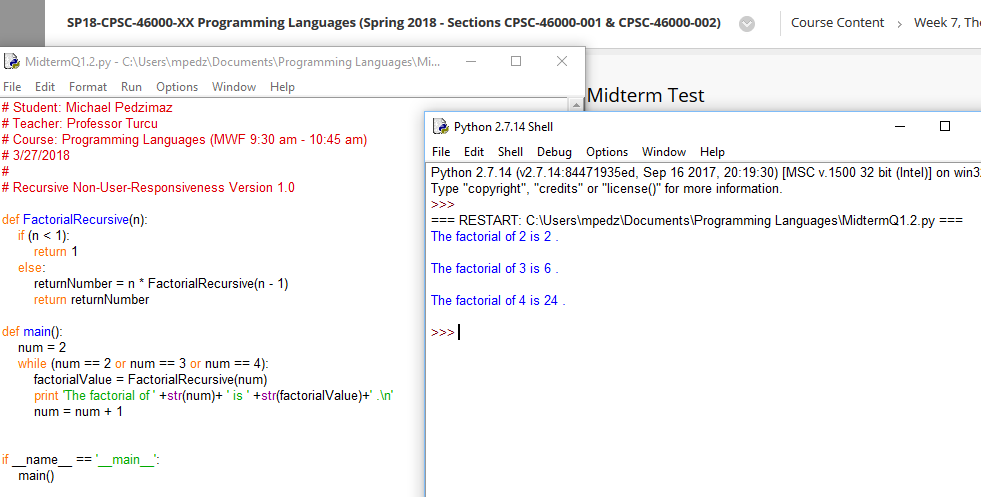
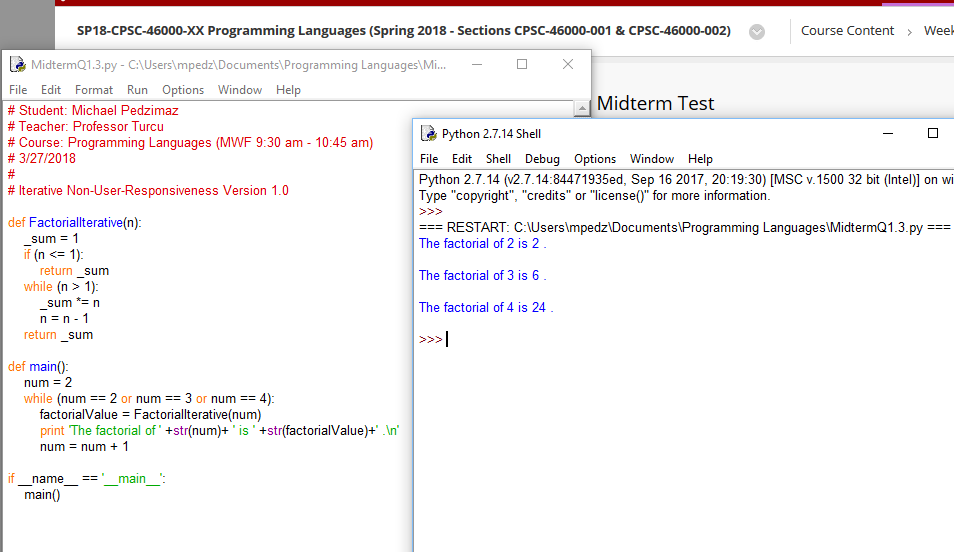
Michael Pedzimaz

**Recursive & Iterative Test Results (Q1)**





**Compilation Process (Q2)**

The compiler implementation is one of three ways that a programming language can be implemented into a computer system, with the other two methods being “Pure Implementation” & a hybrid method of the two. Using the compilation process, programs in a specific language are translated into machine language through a series of steps to run directly on the system. This method has the benefits of very fast program execution (once the translation process is completed) as well as the optional feature of optimization. The first step is to take the *source language*, whether the source code is in C, C++, or any other compiled language, and running it through a lexical analyzer. The LA gathers characters of a source program into something called lexical units, which can be various entities such as operators, punctuation symbols, special words, or identifiers. Comments are, of course, ignored within the lexical analysis. The next step in the process is another analyzer, called the syntax analyzer, taking the lexical units produced by the LA and constructing a hierarchal tree called a *parse tree* out of them. The next step involves something called an *intermediate code generator*, which is fed the previously constructed parse trees, resulting in an intermediate language that lies between the source program & the final output of the compiler; the machine language program. This machine language program is often basically assembly language, while within other situations is still somewhat higher than assembly language. There is also a component called a *semantic analyzer*, which is a part of the intermediate code generator and performs the duty of checking for errors, such as type errors, that were not able to be caught during syntax analysis. Around this part of the compilation process is where optimization (making the program smaller, faster, or both) becomes possible, albeit being unavailable in many situations. Something important to note is that most optimization is done on the intermediate code.

Our next big step in the compilation process is the utilization of a *code generator*. It effectively turns the intermediate code generated by the intermediate code generator into an equivalent machine language program. One very important piece of this puzzle to mention here is that both the lexical & syntax analyzers participated in populating something called a *symbol table*, which serves as a database for the entire compilation process. This table contains essential descriptive information for each user-defined name in the course code. Information such as the type and attribute information are just some examples. The syntax table is used by the code generator & the semantic analyzer.

Another important procedure within the compilation process is the ability to run other types of code along with the machine code produced from the previously detailed steps. The compiler solves this necessity by building calls to required system programs when they are needed, preemptively finding the required programs and linking them to the user program at hand. This is where it gets a little complicated, as the linking operation connects the user program to the system programs by placing the addresses of the entry points of the system programs in the calls to them in the user program. That sentence makes much more sense the more times you read it over! This combined version of the user program & system code are often called a *load module* or *executable image*. This wholes system of linking between the user / system code is called *linking and loading*, which is done by, well wouldn’t you know, the *linker*. Here we arrive at a bottlenecking problem referred to as the *von Neumann bottleneck*. This issue refers to the speed of CPU computations being limited to how fast they can arrive from memory, since the speed of instruction execution is much faster than movement of data into / out of a processor for the actual execution.

**Difference Between Interpreters & Compilers (Q3)**

Interpreters are programs that directly interpret user programs one statement at a time, with not translation whatsoever. This interpreter program acts as a software simulation whose fetch-execute cycle deals with high-level language problem statements rather than machine instructions (Word-for-word from the book, because I couldn’t have said it better myself). One of the main differences between the interpreter method & the compilation method is that the latter is usually 10 to 100 times faster in execution. This is due to the decoding of high-level language statements being much slower than instructions in equivalent machine code. Another difference is that it is easier to implement debugging operations in an interpreted sphere, because all run-time errors could refer directly to source-level units. Another difference is that within interpretation, the program interpreter must decode each statement every single time, causing a performance bottleneck in this area, unlike within a compiled atmosphere, where an entire piece of code is dissected and decoded as one giant unit. There is also a difference in space requirement. Pure interpretation must have both the source code & symbol table present during interpretation.