ELEN513 Final Exam Laurence Kim

Part 1:

The first problem of the exam asked us to write code that could parse a simple code, generate a simple data flow graph with a corresponding, intermediate representation output file. My code takes the instruction code from a TXT file as an input which is shown at the top of the code in p1.py. Once I read from the instruction list file, I initialized an empty list to store my intermediate representation generation. I parse through the instruction file by using regular expression methods to classify the type of instructions that was in each read from the text file. The type of regular expressions that I used were pattern, assign, pattern bin operation, pattern bin integer operation, pattern load, pattern store. Furthermore, throughout the later construction of this code, I created regular expressions for each type of operation, but was swayed to take a simpler route. You will see the other operations as shown that are commented out.

Now to delve into my intermediate representation generation, I read in each Line from the instruction txt file with a corresponding index and appended each instruction in a list that contained relevant information. For the operations, my pattern or format of the list were indices, operations, then registers. For my load operations, the patterns were obviously different from the operations and the order in which the list was stored was index, a string load, expression, null object, and memory address. The store format that was stored into the intermediate representation generation list was index, string store, source, null object, destination. I was able to check the validity of this code by double checking each output of the intermediate representation generation list through prints before I moved onto the graph generation.

My tactic after creating this list was to dump it on a json file which would be very digestible for part two. Furthermore, this problem would not be complete without defining the data flow graph, which I did so using graphviz. I had many different starting points for this problem, but settled on the one that I submitted in the p1.py file. My graph generation handles dependencies properly. I iterate throughout the entire intermediate generation list and start by creating a node in order that is read from the IR . As I iterate through the ir, creating nodes for each instruction, I traverse backwards to the most recent instruction all the way to the first instruction to see if any previous instructions use the registers that will be utilized in the current operation. If the Code finds a match, then it will create an edge from the current instruction that is being generated to the most previous instruction that was generated as a node. Furthermore, the store operation only needs to check for one register to find a dependency. In the plethora of tests, I try to break this code which I was able to do countless times. I was able to settle on a parser that I feel confident in.

Part 2:

My vision for this code was to create my own directed graph that is composed of a list of nodes and a list of edges. Reading in from the json file, I appended the according latency to each instruction based on the value that was defined. I also have a variable that defines the number of PEs, which is the goal of this problem. My vision was to properly load all the instructions from the intermediate representation json file and prune or merge, depending on ease. My initial vision consisted of several principles as to create the most optimal, merging of each node down to the number of PEs that is defined. One of the principles that I considered was an outside in reduction rather than a simple greedy approach. Meaning I would combine

the highest value latency PE with the most probable low latency value node. Another approach I considered was an inside out merging which would have proven to be interesting. I refrain from simply pruning the tail nodes, which did not have any dependencies towards it, as it would not be the most significant indicator of finding the most optimal solution. The approach that I attempted was to merge the nodes with its most immediate upper dependency until the expected number of pes was reached. I was able to find a solution that did this not quite to fruition, but almost all the way and I would've generated it in a digraph, which is shown in P2 to double check the correct output.

Finally, I did not have enough time to finish p3 based on p2, however I have some pseudo code based on my parser from early on in my works.