We tried to implement the following steps to generate IL code from the given grammar:

1. Generate a binary syntax summary tree using the Listener, input, and grammar.
2. Convert the binary syntax summary tree to a regular tree.
3. Convert the constructed tree to a graph in the networkx library.
4. Post-order traversal of the Abstract Syntax tree by networkx and save the traversal order.
5. Generate IL code from the post-order traversal and produce IL

The process of running the program is as follows: First, in the main script, we take the input and then convert it into a binary syntax summary tree using the CustomASTListener class.

if \_\_name\_\_ == '\_\_main\_\_':  
 argparser = argparse.ArgumentParser()  
 argparser.add\_argument(  
 '-n', '--file',  
 help='Input source', default=r'input2.txt')  
 args = argparser.parse\_args()  
 main(args)

ast\_generator = ASTListener()  
# Step 7(a): Walk parse tree with a customized listener (Automatically)  
walker = ParseTreeWalker()  
walker.walk(t=parse\_tree, listener=ast\_generator)

Then, using the transform\_binary\_ast\_to\_normal\_ast function defined in the ast script, we convert the binary tree into a regular tree.

normal\_ast = transform\_binary\_ast\_to\_normal\_ast(ast\_generator.ast)

Then, using the transform\_ast\_to\_networkx function defined in the ast\_to\_networkx\_graph.py script, we import the tree into the networkx environment.

graph = transform\_ast\_to\_networkx(normal\_ast.root)

Then, we traverse the tree in post-order and pass the result to the generate\_intermediate\_language function defined in the ILMapper class to generate the IL code and save it in the output.il file.

s = 0 # choose a source node  
stack =[]  
for node in nx.dfs\_postorder\_nodes(graph, source=s): # iterate over nodes in postorder  
 label = graph.nodes[node].get('label', 'No label')  
 stack.append(label) # display the node  
il\_mapper = ILMapper()  
print(il\_mapper.generate\_intermediate\_language(stack))

The generate\_intermediate\_language function actually generates intermediate code or IL code by receiving the post-order traversal created in the previous steps. For this purpose, it traverses the received stack and wherever it reaches an operand, it stores it in the stack, and wherever it reaches an operator, it calls the generate\_il\_based\_on\_operator function. Based on the type of input operator, it pops the corresponding number of operands from the stack and generates its IL code. The generated codes are stored in a list named il\_codes, which can be used later in the development and completion of the program.

def generate\_intermediate\_language(self, post\_order\_array, pre\_order\_array=None):  
 for item in post\_order\_array:  
 if self.is\_operator(item):  
 self.il\_codes.append(self.generate\_il\_based\_on\_operator(item))  
 else:  
 if self.is\_identifier(item):  
 self.add\_global\_variable(item)  
 self.stack.append(item)

The generate\_il\_based\_on\_operator function, as explained above, pops as many operands as the input operator requires from the stack based on the operator it receives as input. It then generates and returns the IL code related to that operator.

def generate\_il\_based\_on\_operator(self, item):  
 if item == '\":=\"':  
 second\_operand = self.stack.pop()  
 first\_operand = self.stack.pop()  
 return self.assignment(first\_operand, second\_operand)  
 if item in self.arithmetic\_operators:  
 second\_operand = self.stack.pop()  
 first\_operand = self.stack.pop()  
 return self.arithmetic(first\_operand, second\_operand,item)  
 if item in self.relational\_operators:  
 second\_operand = self.stack.pop()  
 first\_operand = self.stack.pop()  
 return self.relational(first\_operand, second\_operand, item)  
 if item == '\"?:\"':  
 second\_operand = self.stack.pop()  
 first\_operand = self.stack.pop()  
 condition = self.stack.pop()  
 return self.ternary(condition,first\_operand,second\_operand)  
 if item == 'if':  
 pass

**Exercise**: Modify the given code to add the ability to generate IL code for IF, FOR, WHILE, and compound statements to the program.

**Additional points**: In the defined grammar, there are statements that are not just one line and include a block of code (like if, while commands, etc.). Since the scope of these types of sentences is not recognizable after traversing the AST tree, it is not possible to implement their commands with the current process. You can change the AST tree and specify the beginning and end of multi-line statements, or customize the post-order traversal of the tree, determine the start and end of the statements, and then proceed to generate their IL code.

**The deadline for submission is until midnight on December 22, 2023.**