# Project 1 Documentation:

# Recursive Descent Parsers

Vincent Testagrossa

University of Maryland Global Campus

CMSC 330: Advanced Programming Languages

Dr. Yuhua Jiang

September 11th , 2022

# Contents

## [Testing](#_Testing)

## [Reflection](#_Reflection)

# Testing

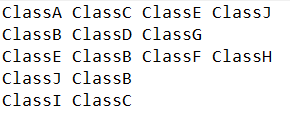
## Testing Provided Calculator Sample:

Input:

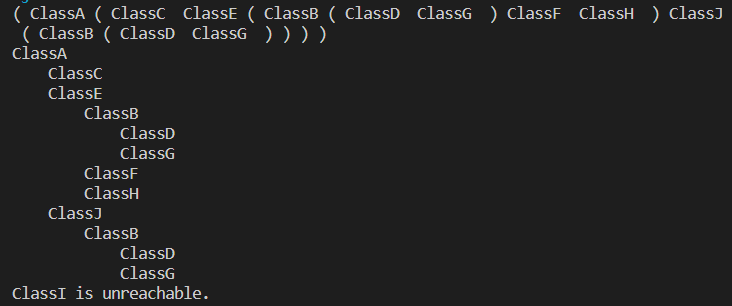
Output:

## Testing with No Circular Dependencies

Input:

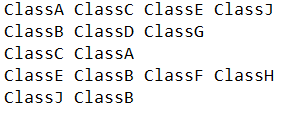


Output:

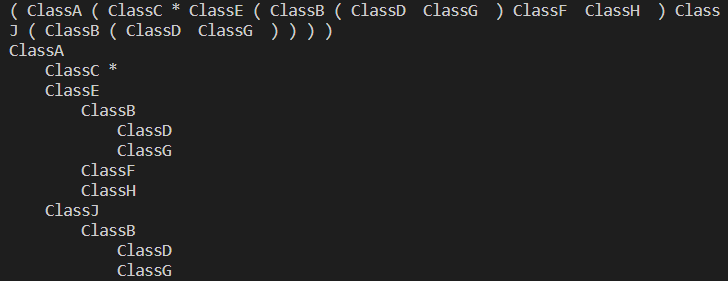


## Testing with No Unreachable Classes

Input:



Output:



# Reflection

Initially I had started by using a HashMap to create the list of vertices, with the linked lists as the adjacency list, but I figured that might be “cheating” a bit, since the HashMap isn’t strictly a LinkedList, as figure 10.7 described in the text. Using the built-in Java version of the LinkedList was a lot easier, since I didn’t have to implement a lot of the data structure for that particular ADS, and all I really had to worry about was creating the specialized Vertex node and use that for the linked lists. I went in a different direction from the examples and included “discovered” and “visited” flags as part of the structure of a Vertex, instead of creating lists of each (since the vertices were already stored in a list) like I was seeing in a lot of other examples. I also associated the adjacency lists directly with each Vertex as another new LinkedList<Vertex> so that everything associated with a depth-first search can be done by accessing the first Vertex. I was able to forgo another separate list of edges by doing it this way, along with an array or list of “marks”. The biggest issues I had were with the parenthesized list and hierarchy both showing the behavior of the algorithm descending and ascending into vertices that it didn’t necessarily need to. I chose to account for this in each of those classes with some formatting tricks, rather than try and fix every edge case within the required algorithm.