CSE 262: Quiz #1  
Due September 16th, 2022 at 11:59 PM

The quiz has TWO questions. Please submit your answer by updating this file in the quizzes folder of your Bitbucket account, and then committing and pushing. You should use as much space as you want for each answer. Please be detailed in your answers. Remember: this quiz is worth 9% of your grade, and you will not receive very many points if you do not give detailed answers.

**Question 1:** Extend the grammar from Figure 2.25 with an exponent operator (^) that has higher precedence than multiplication and division, and with a modulus operator (%) that has lower precedence than multiplication and division, but higher precedence than addition and subtraction. Accompany your proposed grammar with text explaining what you did and why.

1. program −→ stmt list $$ // this is the new and extended proposed grammar
2. stmt list −→ stmt list stmt // the highlighted parts are what ive added and changed
3. stmt list −→ stmt // below this grammar is my explanation and
4. stmt −→ id := expr
5. stmt −→ read id
6. stmt −→ write expr
7. expr −→ bar
8. expr −→ expr add\_op bar
9. bar −→ term
10. bar −→ bar mod\_op term
11. term −→ foo
12. term −→ term mult\_op foo
13. foo −→ factor
14. foo −→ foo exp\_op factor
15. factor −→ ( expr )
16. factor −→ id
17. factor −→ number
18. add op −→ +
19. add op −→ -
20. mod\_op −→ %
21. mult\_op −→ \*
22. mult\_op −→ /
23. exp\_op −→ ^

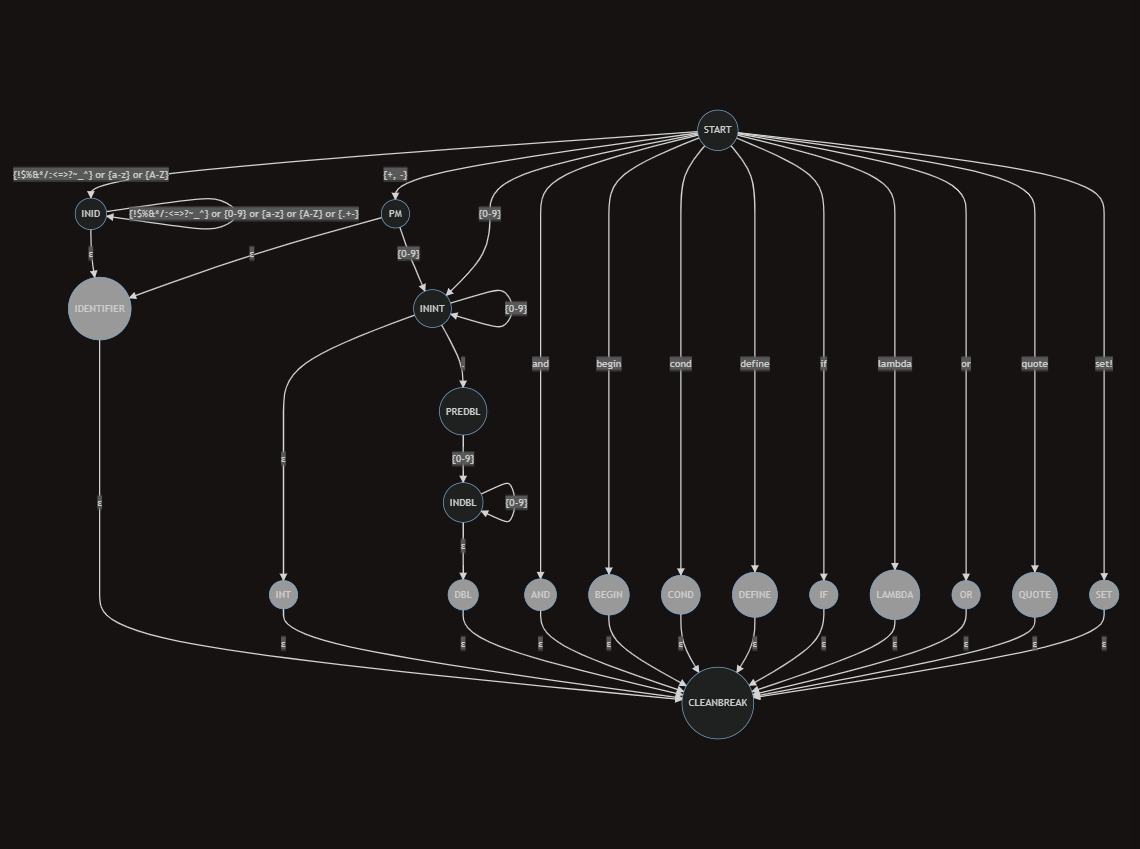
In my new proposed grammar, I made two new symbols called “bar” and “foo”. The bar symbol’s production is meant for the modulus operation (%) and the foo symbol’s production is meant for the exponential operation (^). This grammar before being extended was already written in left recursion (as indicated by the recursive call being on the left in the production of symbols “expr” and “term”), and so I followed the left recursive writing (as indicated by the recursive call also being on the left in the production of the new symbols “bar” and “foo”). In addition, left recursion is good because of reduction and left associativity (evaluated left to right precedence for same type of operators).

To explain the order in which I made my grammar to give it its respective precedence, I would like to explain the parsing tree first. The parsing tree executes from the leaves of the tree (from left to right) up to the root of tree. Therefore, for the respective operation to have a higher precedence than another operation, it would have to be on a lower level in the tree (towards the leaves) than the level of an operation with lower precedence. Therefore, for the exponential operation to have a higher precedence than multiplication and division, the symbol; “foo” had to be on a lower level in the tree than the symbol “term”. That also means that for the modulus operation to have lower precedence than multiplication and division, “bar” had to be on a higher level in the tree than the symbol “term”.

The execution of the non-terminals (assuming each unique symbol is executed) goes from program- > stmt -> expr -> bar -> term - > foo -> factor (from highest level to lowest level of tree) which shows that my new grammar does have the correct level of precedence. Furthermore, in the production of “exp”, “bar”, “term”, and “foo”, they recursively call themselves with their own non-terminal. Therefore, multiple of their own operations are still being done at higher levels in the tree than the operations with higher precedence which should be at a lower level in the tree. (Additional note, taking either the top to bottom approach or bottom-up approach does not matter as it would form the same tree.)

**Question 2:** In your repository’s p1 folder, you will find the files `scanner\_ids\_nums.png` and `scanner\_keywords.png`. When we combine them (by merging the common states), the result is a nondeterministic finite automata. First, you should explain why it is nondeterministic. Then, you should create a new diagram consisting of the composition of the two automata, but with whatever changes would be needed in order to make it deterministic. Be sure to explain what you did, and why. Is your construction *minimal*?

Merged the common states:



A deterministic finite automaton knows what state to transition to based on a given input. Therefore, there may not be two transitions from a given state for the same input and there also may not be epsilon transitions. Merging the common states of `scanner\_ids\_nums.png` and `scanner\_keywords.png` results in a nondeterministic finite automata because it has epsilon transitions and certain states are trapped because they do not have a transition for every single possible input.

Next page is my Deterministic Finite Automation: (Below)

DFA combined:

A picture containing text

Description automatically generated

The black colored states are accepting states. The white colored states are just transient and non-accepting states.

ACCEPTING STATES: the special formed words, IDENTIFIER, INT, DBL.

NON ACCEPTING STATES: START, INID, PM, PREDBL

Whitespace input = {‘ ‘, ‘\t’,’\n’,’\r’}

This is a Deterministic finite automation because each of the states have a transition for every possible expected given input. If the transition for an input is not shown in a state above, it is trapped, and it should throw an error. No singular state has two transitions for the same input and no state has an epsilon transition. In the START state, it can either transition to the INT state if the input was {0-9}, or the PM state if the input was {+ -}, or the INID state if the input was anything else.

I made a major change to the INID state because we had to incorporate the special form words. If the input is the first letter of any of the special form words, it goes through all of the states for the word that starts with the inputted letter, checking if the special form word was the exact word that was formed and each character is in order

(EX: “and” cannot be in “nad” or “andp”). (I did not include the states for each special form word character in the graph to simplify it). If the special form word was found, it will transition to the specific special formed word state, otherwise If it catches any extra letter, or letter not in the correct position, it will transition to the IDENTIFIERE state. Once in the special formed word state, it can either transition to the START state if given a white space input, indicating a new token, or it can transition into the IDENTIFIER state given any other input since any additional input will make it an IDENTIFIER and not a special formed word. In the IDENTIFIER state, the IDENTIFIER will continue to call itself and append any input until a white space input, indicating a new token, where it will transition to the START state.

Another major change I made was removing the ININT and the INDBL state. The START state now transitions directly into the INT state if the input is a number. If START state was {+ -}, it transitions into the PM State. In the PM state if the next input is {0-9} it will move to the INT state as a signed INT, otherwise it will move into the IDENTIFIER state. The reason why I removed the ININT state was because if the input is a digit {0-9} than it has to be an int, and it will continue to call itself in the accepting state, appending as many digits as wants and still be an INT. The accepting INT state now also transitions to the PREDBL state If the input is a ‘. ‘, but transitions into the START state if it is a white space, indicating a new token. The PREDBL state now transitions into the DBL state, instead of the INDBL state, if the next input is a number. The reason why I removed the INDBL state is because if the INT has a ‘.’ followed by a digit, then it must be a double. DBL will continue to call itself, appending as many digits as it wants until a whitespace which indicates a new token.

As stated in the beginning, if the transition for an input is not shown in a state above, it is trapped, and it should throw an error. Normally for an DFA we should show trap states, but I did not show it here because it will make it too hard to see the states in a cropped picture. (In the same quiz directory I also have a PDF and JPG of the DFA if its too hard to see). I would also like to note that I removed the CLEANBREAK state because for each of the accepting states I added a transition to the start when the next input is a white space indicating a new token.

I do not think that this DFA is fully minimalized. The DFA’s transient states are: START, INID, PM, PREDBL, and the small states between checking for the special form words. The rest of the states are the accepting states. In the transient states, PM could have transition to the same state with the same value as the START state, which makes it not minimalized. Furthermore, I think it is redundant to transition to the IDENTIFIER state if any of the characters are not in the correct letter or in the correct spot for the small states checking for the special formed word.