Caleb Klenda
Homework 2
2/25/2025

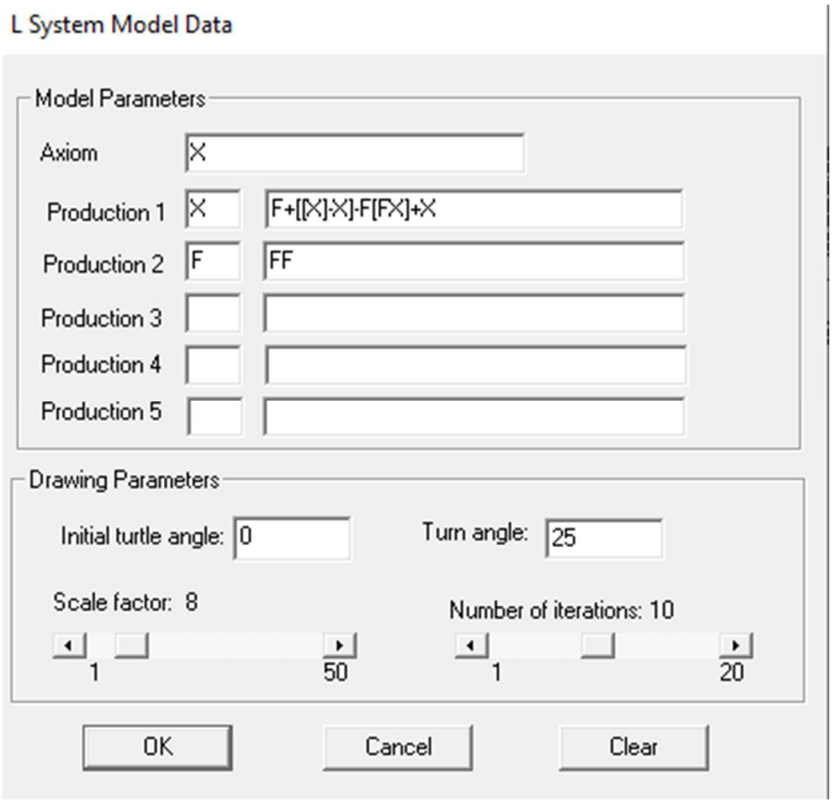
# Problem 1

This first model I call the “forever tree” since it looks like a laid down tree cut in half. It leverages the use of brackets to create a branching fractal. The definition is as follows:

- **Alphabet**: {F, X, +, -}
- **Axiom**: "X"
- **Rules**:
    - X → F+[[X]-X]-F[-FX]+X
    - F → FF

“X” operates as a placeholder for the branching expansions. The brackets allow for recursive branching, where:

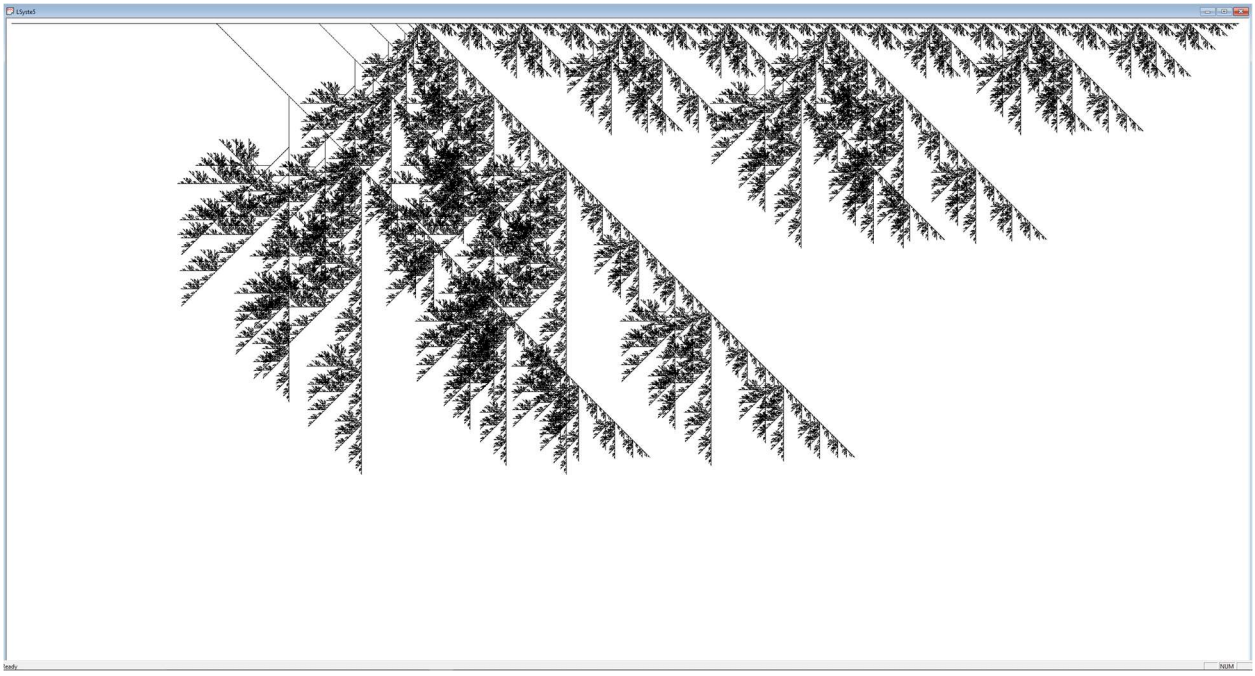
- “[“pushes the position and angle onto a stack.
- “]” pops the last position and angle, returning to that point.



Running the system with an angle 25°:

With 35°:

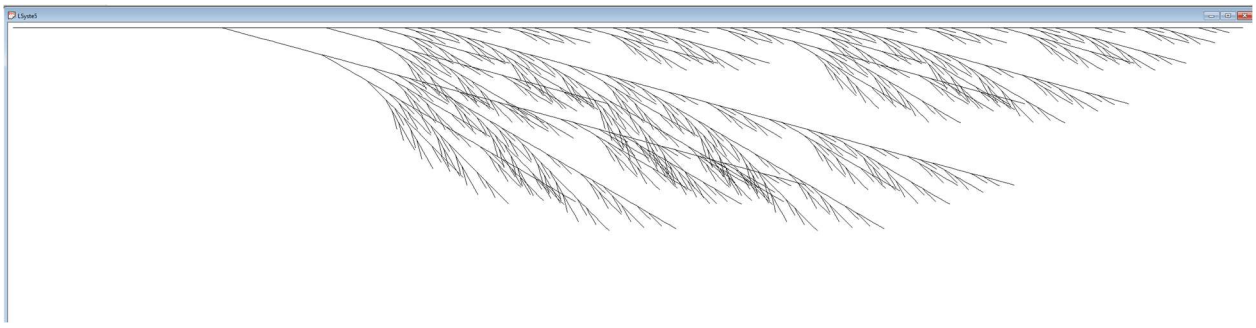
With 45°:



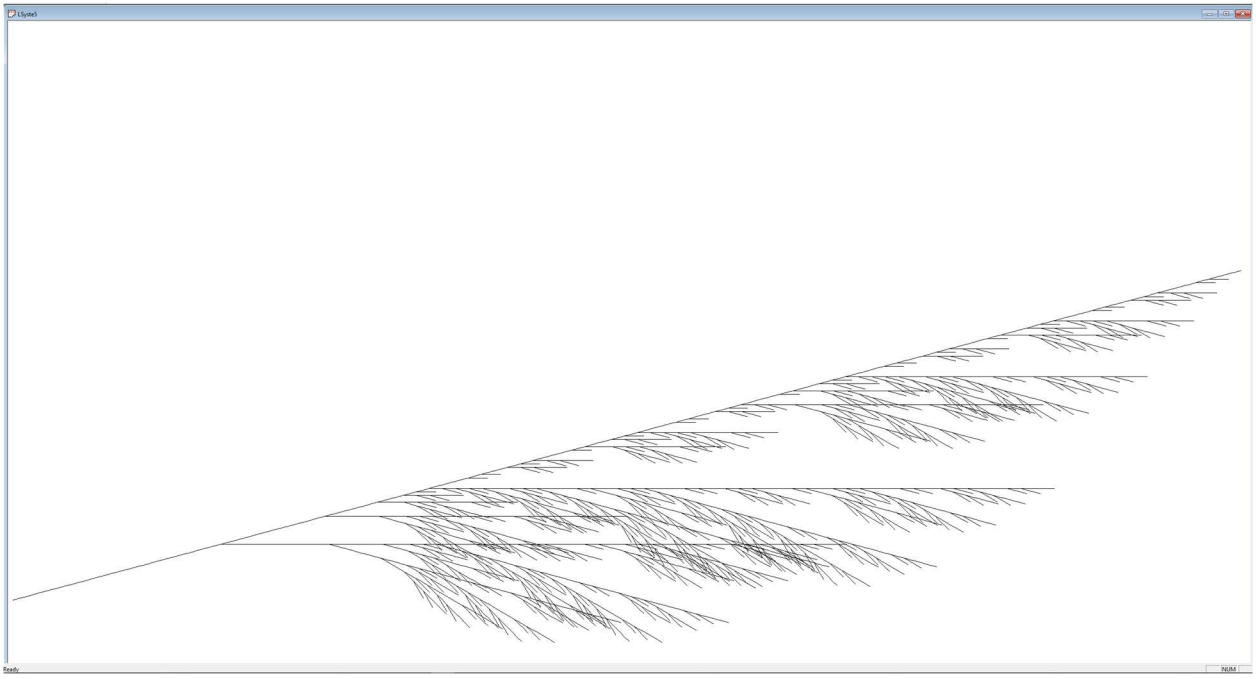
The shallow angles make the image more tree-like whereas the larger ones makes the image more crystalline.

Bumping back down the angle to 15 and lowering the iterations results in a tree-like structure that is less dense. Changing up the initial angle allows for a different placement of the base stem of the branch.

With initial angle 0°, 6 iterations, 15° turn:



With initial angle 15°, 6 iterations, 15° turn:



Other notes from experiment are the complex production rules do not always generate complex and beautiful fractals. It took quite a bit of experimentation to get something branch like without devolving into a chaotic mess.

# Problem 2

2.1)

- The non-terminal symbols are: E, T, & F

- The terminal symbols are: +, -, *, /, (, ), ident, & number

2.1) Derving id-id-id with G2 (left-most)

**Start with E**
→ E - T (using E → E - T)

**Expand the leftmost E**
→ (E - T) - T (using E → E - T)

**Expand the leftmost E**
→ (T - T) - T (using E → T)

**Expand the leftmost T**
→ (F - T) - T (using T → F)

**Expand the leftmost F**
→ (id - T) - T (using F → id)

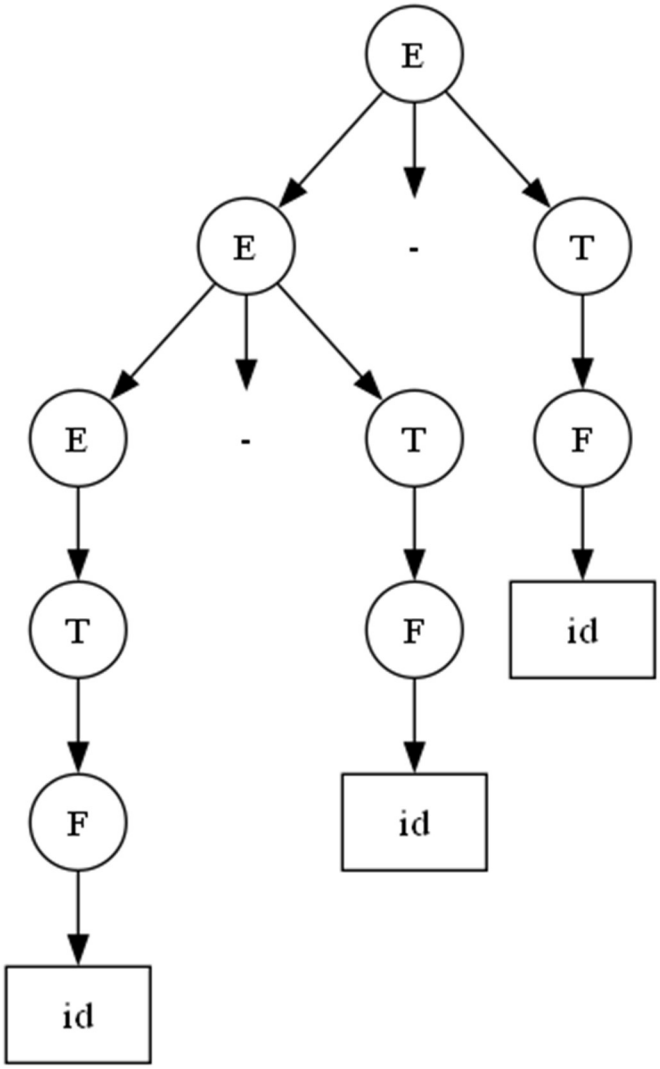
**Expand the next T**
→ (id - F) - T (using T → F)

**Expand the next F**
→ (id - id) - T (using F → id)

**Expand the last T**
→ (id - id) - F (using T → F)

**Expand the last F**
→ (id - id) - id (using F → id)

This results in the parse tree on the right:



2.3)

**Start with E**
→ E - T (using E → E - T)

**Expand the leftmost E**
→ (T - T) (using E → T)

**Expand the leftmost T**
→ (F - T) (using T → F)

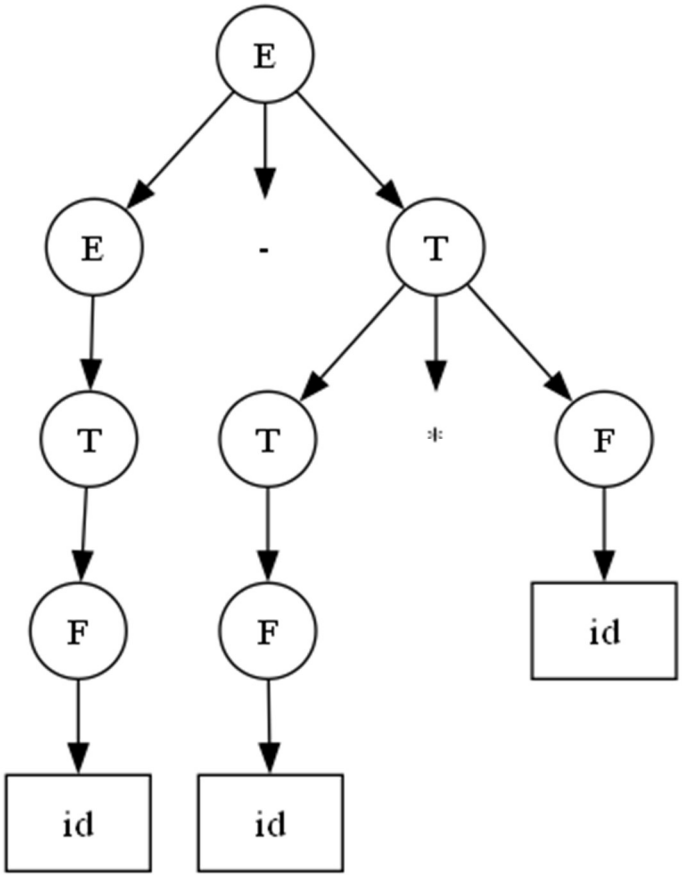
**Expand the leftmost F**
→ (id - T) (using F → id)

**Expand the next T**
→ (id - T * F) (using T → T * F)

**Expand the leftmost T in T * F**
→ (id - F * F) (using T → F)

**Expand the first F in F * F**
→ (id - id * F) (using F → id)

**Expand the last F**
→ (id - id * id) (using F → id)

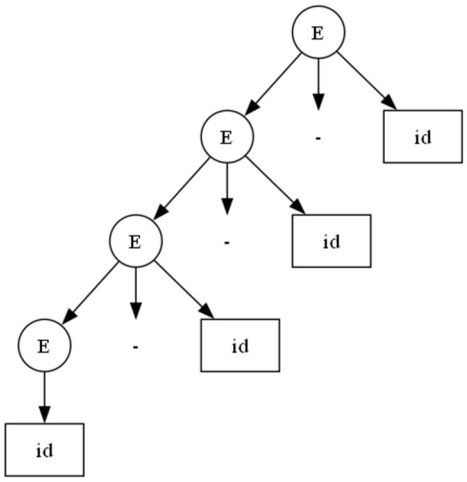


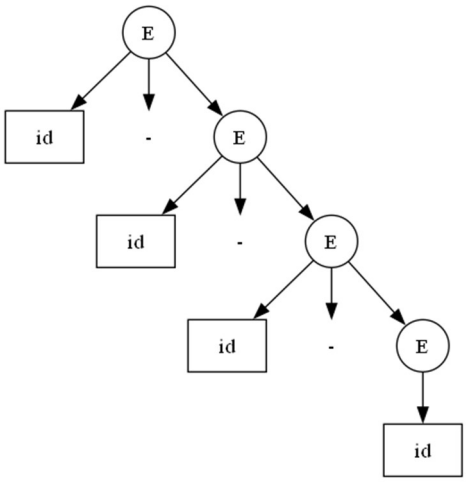

2.4)

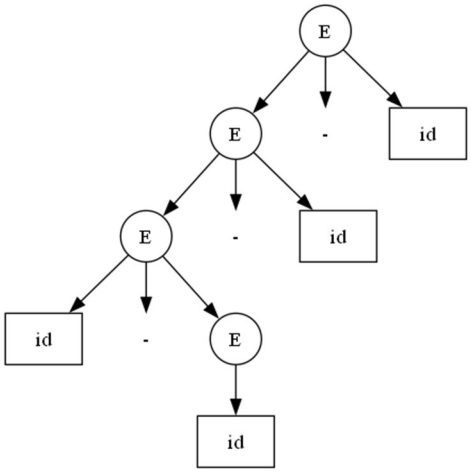
For “id-id-id”, the subtraction operator results in the following order of evaluation: Compute id - id first; Then, take the result and subtract id. This doesn’t matter much since subtraction is commutative with itself. However, for “id-id*id” multiplication and subtraction are not communicative with one another and so the order of operations matters. In this case, G2 will enforce such an order by ensuring that id*id happens before id – (id*id).

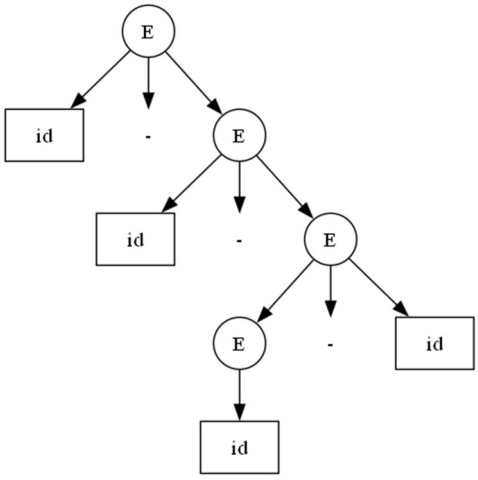
# Problem 3

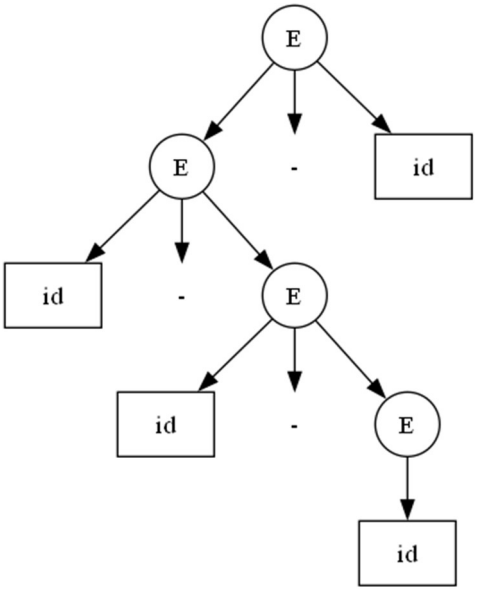
3.1) Since it's ambiguous, multiple trees can exist; a total of 6 are possible.

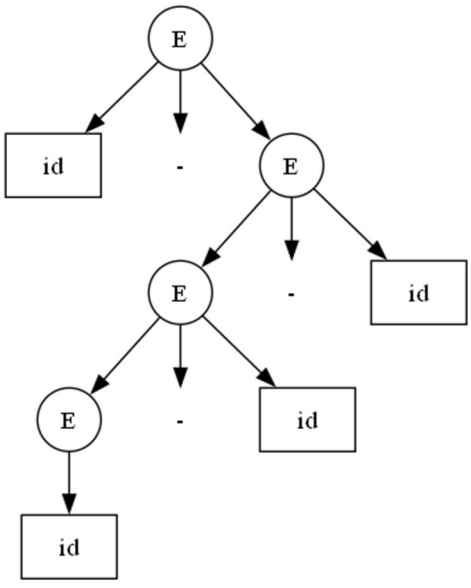


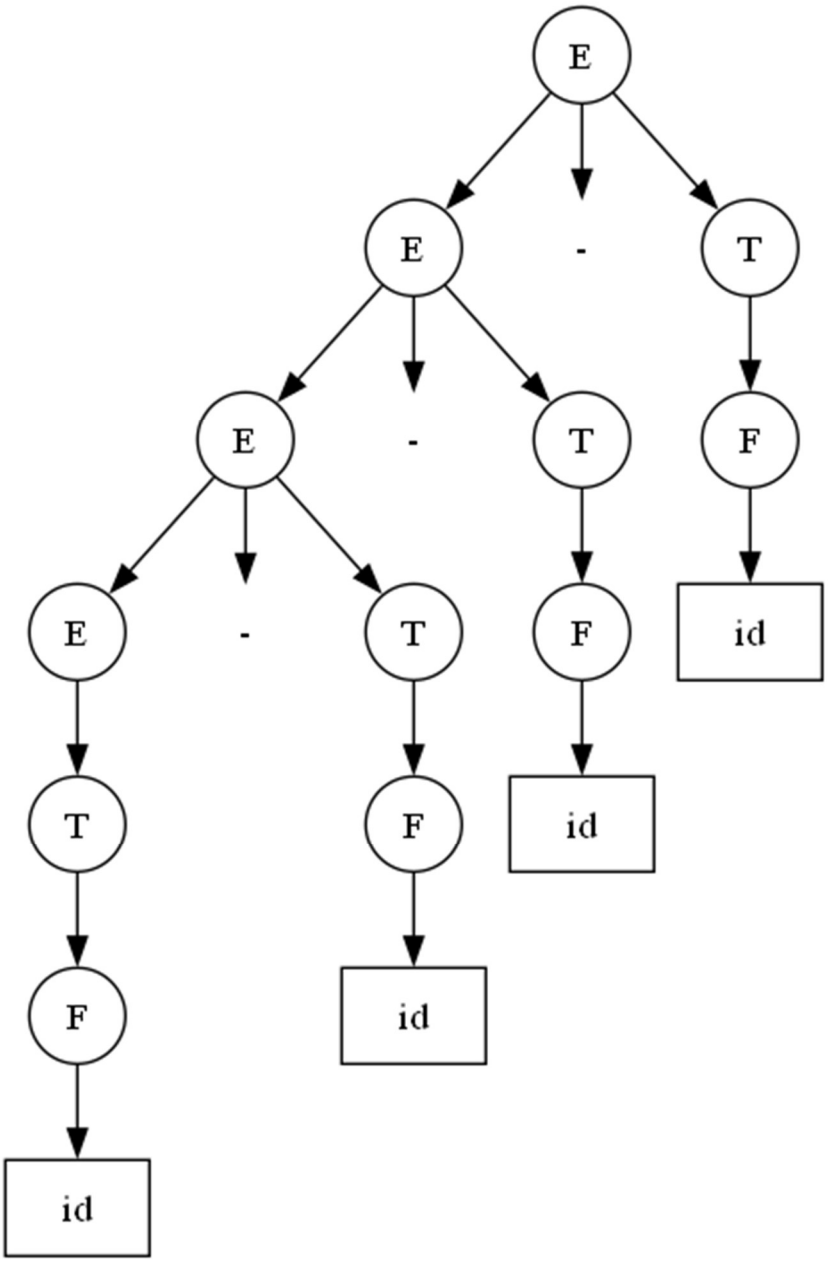



3.2) Only one tree can be constructed since it is fully left associative with the operators.



3.3)

**Start with E**

→ E + T (using E → T { ( ‘+’ | ‘-‘ ) T })

**Expand E**

→ E - T + T (using E → T { ( ‘+’ | ‘-‘ ) T })

**Expand E**

→ E - T - T +T (using E → T { ( ‘+’ | ‘-‘ ) T })

**Expand E**

→ T - T – T + T (using E → T { ( ‘+’ | ‘-‘ ) T })

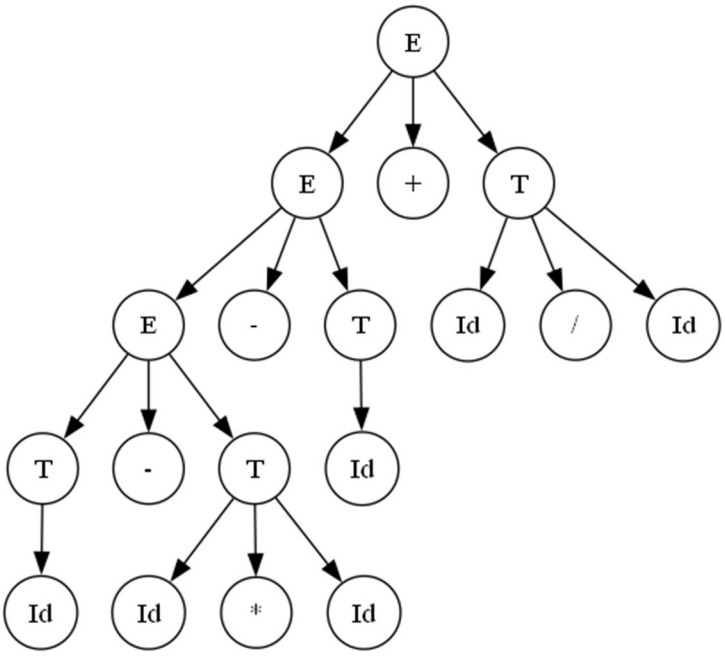
**Expand T**

→ F - F * F - F + F / F (using T → F { ( ‘*’ | ‘/‘ ) F })

**Expand F**

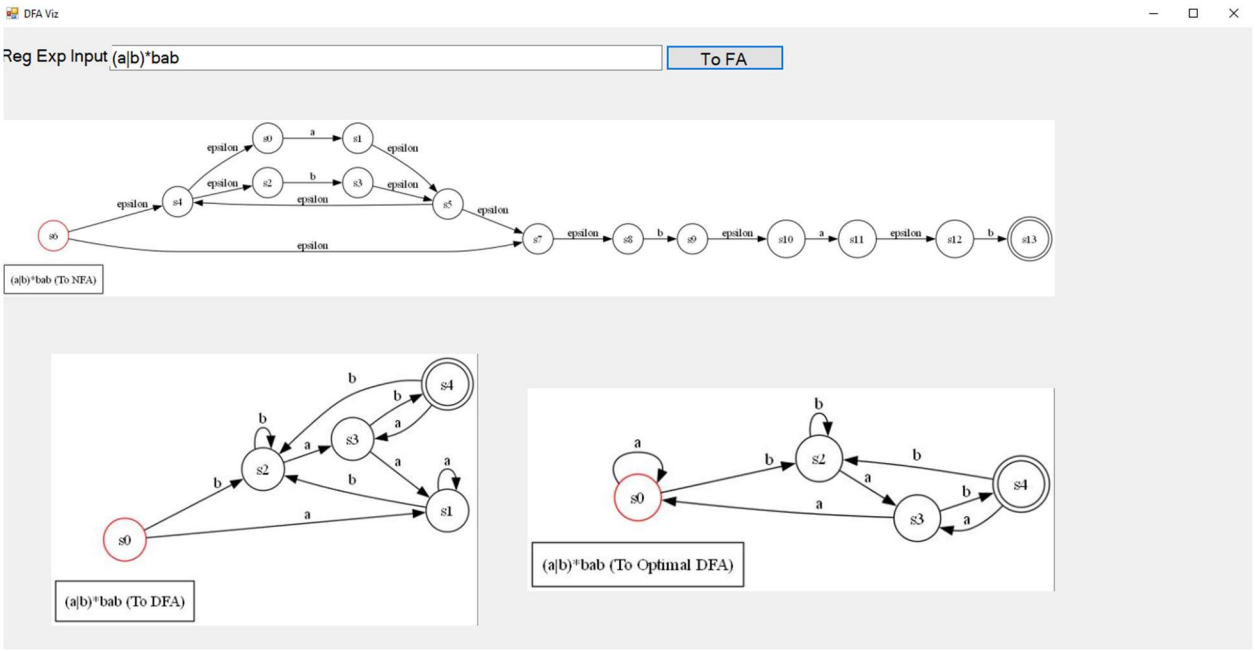
→ id – id * id – id + id / id

(using F → ‘(‘ E ‘)’ | ident | number)



# Problem 4

4.1)




4.2) To determine if the string “abbabab” is accepted, we trace through the optimal DFA given above. If a path to the ending state S4 is found, then the string is accepted by the regular expression.

We can see the following path does accept the string: s0→s0→s2→s2→s3→s4→s3→s4→done

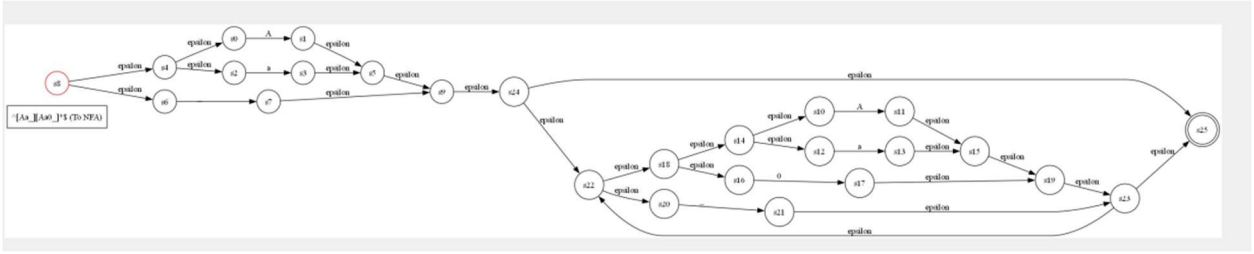
4.3) This rather esoteric string gives us the regex for C identifiers:

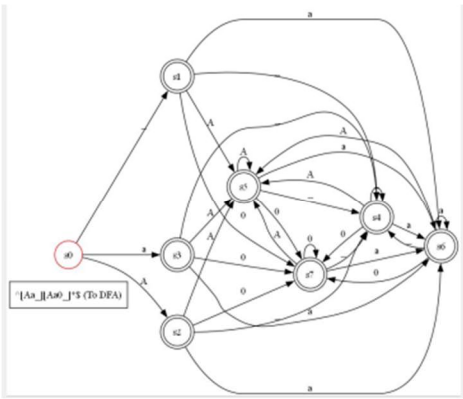
*^[a-zA-Z_][a-zA-Z0-9_]*$*

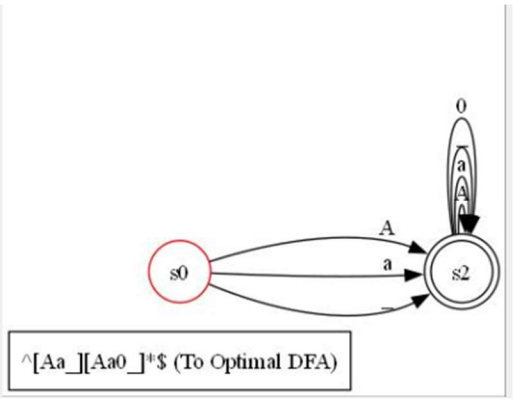
This ensures that every identifier starts with an” _” or character. This would produce too many state for DFAViz so we will abstract the lower case letters as “a”, the upper case letters as “A” and the numbers as “0”. This yields:

*^[Aa_][Aa0_]*$*

Which produces this NFA and DFA:

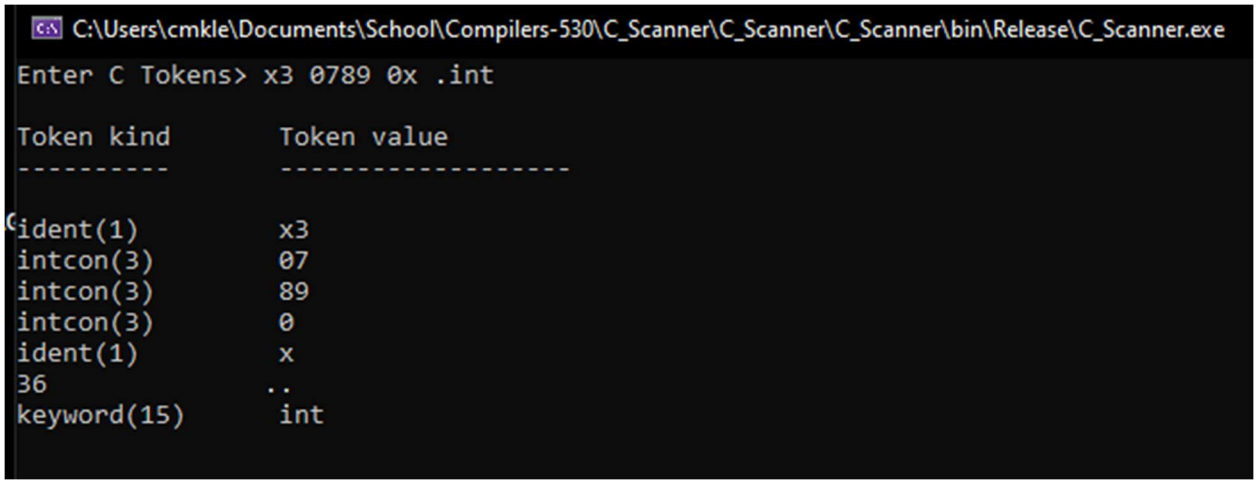



s0
s1
s2
s3
s4
s5
s6
s7
^[Aa_][Aa0_]*$ (To DFA)


s0
s2
A
a
0
^[Aa_][Aa0_]*$ (To Optimal DFA)

# Problem 5

5.1)




**"0789"** starts with 0, making it a valid octal integer literal in C. However, 8 and 9 are invalid digits in octal it treats those separately. The dot (.) usually indicates a floating-point number if followed by digits. However, here it's followed by a non-digit (i), which means the scanner instead reads it as its Unicode value (how it gets the 36).

5.2)

The greedy method in a scanner refers to the strategy where the scanner or lexer always matches the longest possible valid token from the input stream. This ensures that the lexer does not prematurely break tokens into smaller pieces when a longer valid token exists.**:**

**Example 1:**

intvalue = 10;

The lexer needs to decide whether "intvalue" is the keyword int or an identifier. Using the greedy method, it scans forward and finds that "intvalue" is not a reserved keyword, but rather an identifier.

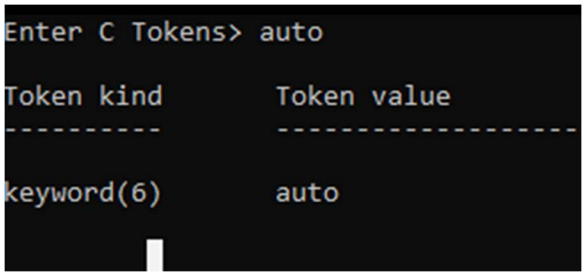
**Example 2:**

a = b +++ c;

The lexer needs to decide between: ++ (increment) and + (addition) or + (addition) and ++ (increment). Using the greedy method, it first matches "++" (longest match, increment operator), then, it matches "+" separately.

5.3)

”auto” matches a keyword and an Identifier as token type. However, it returns as keyword since the scanner values the precedence of keywords over identifiers.




5.4)

The C_Scanner program determines input type by using Scanner(Stream s) or a file Scanner(string filename) or keyboard or file input respectively. Then to handle both ASCII & UTF-8, the scanner assumes the input is a byte stream and interprets it based on character definitions in CHARACTERS. The scanner will analyze the byte sequence at the start of the input: ASCII characters fall within the 7-bit range (0–127) and are compatible with UTF-8. UTF-8 uses multi-byte sequences for non-ASCII characters. If the input contains only bytes 0–127, it is likely ASCII. If bytes beyond 127 appear in valid UTF-8 sequences, the scanner recognizes UTF-8.