**CS 4280, Section 001, Academic Integrity Statement**

For this project, AI resources can be freely used, as long as they are fully disclosed as described herein. Additional non-AI internet resources can be utilized, as long as they are fully disclosed. Furthermore, code written by UMSL students in ***previous*** semesters can be looked at, but never copied.

**IMPORTANT**: Clearly indicate all outside resources utilized and sign below. Failure to cite the use of outside resources will be reported for appropriate disciplinary actions. Note that discussions with other students are encouraged; looking at each other’s code and/or copying – with or without modifications – are unacceptable and will be reported.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

I hereby certify that all outside resources utilized, other than suggested textbooks and class materials, are clearly noted in my source code and in the following. The start and finish lines of the affected code are shown using ‘start ORx’ and ‘end ORx’, respectively, where ‘x’ is a unique number. Each value of ‘x’ has a corresponding explanation in this Academic Integrity Statement.

All internet resources include the web address and the date accessed. For each usage of AI, I also include the prompt and code submitted and the output returned or a link to the interaction is included herein.

All other materials I provide for this project submission are my own original work. I hereby certify that I am responsible for each and every line of code that I submit in my source files and I thoroughly understand how the code works to produce the output. I understand that I wll be required to meet with the instructor to answer detailed questions about my submitted code.

OR1:

<https://www.programiz.com/cpp-programming/library-function/cctype/isalpha>

* Looking for letter -> token 3 validation
* if (!isalpha(str[0]))
* return false;

OR2:

* <https://docs.vultr.com/cpp/standard-library/cctype/isdigit>
* Looking for digit -> token3 validation
* if (!isdigit(str[i])) *// checking if the rest are digits*
* return false;

OR3:

* https://cplusplus.com/reference/string/string/
* *// function to remove comments*

string removeComments(string str)

{

size\_t startOfComment = str.find('\*');

while (startOfComment != string::npos) *// if another '\*' is found*

{

size\_t endOfComment = str.find('\*', startOfComment + 1);

if (endOfComment == string::npos) *// there is no other '\*'*

break;

return str.substr(0, startOfComment) + str.substr(endOfComment + 1); *// removing the comment and everything in-between*

}

return str;

}

OR4:

* <https://www.w3schools.com/cpp/cpp_enum.asp>
* Use enums to access the type of token
* */\**
* *Author: Grant Hughes*
* *Created: February 23, 2025*
* *token.hpp:*
* *- header file for token*
* *\*/*
* #ifndef \_TOKEN\_HPP\_
* #define \_TOKEN\_HPP\_
* #include <string>
* enum TokenID {
* t1\_tk,
* t2\_tk,
* t3\_tk,
* EOFTk
* };
* struct Token {
* TokenID tokenID;
* std::string tokenInstance;
* int lineNumber;
* Token(TokenID id, const std::string &instance, int line): tokenID(id), tokenInstance(instance), lineNumber(line) {}
* *// Method to get the string representation of the token type instead of int value*
* std::string getTokenAsString() const {
* switch (tokenID) {
* case t1\_tk: return "t1\_tk";
* case t2\_tk: return "t2\_tk";
* case t3\_tk: return "t3\_tk";
* case EOFTk: return "EOFTk";
* default: return "tk\_unkown";
* }
* }
* };
* #endif

OR5:

Use claudeAI to help with an error in code.

User code:

code: \*/ **Author: Grant Hughes Created: March 23, 2025 parser.cpp: \* - Implamentation of parser interface for BNF grammar\*** /\* #include "token.hpp" #include "node.hpp" #include "scanner.hpp" #include "parser.hpp" #include <iostream> #include <vector> #include <string> #include <fstream> using namespace std; // global varaibels to track tokens static vector<Token> tokens; static size\_t tokenSize = 0; // prototypes node\* S(); node\* A(); node\* B(); node\* C(); node\* D(); node\* E(); node\* F(); node\* G(); // helper function to return current token Token& currentToken() { if (tokenSize < tokens.size()) { return tokens[tokenSize]; } \* // otherwise retunr EOF\* static Token EOFToken(EOFTk, "EOF", -1); return EOFToken; } // helper fucntion to return errors in grammer void parsingError(const string& expected) { Token current = currentToken(); cout << "PARSER ERROR: Expected " << expected << ", got '" << current.tokenInstance << "' on line " << current.lineNumber << endl; exit(1); } // return the next token void nextToken() { if (tokenSize < tokens.size()) { tokenSize++; } } // S -> A(BB) node\* S() { node\* nodeForS = new node("S");\* // creating non-terminal S(root)\* nodeForS->addChildren(A());\* // creating A (non-terminal)(node)\* if (currentToken().tokenInstance == "(") { node\* leftParenthesis = new node("t1", "(");\* // creating left parenth\* nodeForS->addChildren(leftParenthesis);\* // adding left penth to tree as child\* nextToken(); } else { parsingError("( in S non-terminal grammar"); } \* // adding the two B non-terminals\* nodeForS->addChildren(B()); nodeForS->addChildren(B()); \* // handing the end parenth (')')\* if (currentToken().tokenInstance == ")") { node\* rightParenthesis = new node("t1", ")");\* // creating right parenth\* nodeForS->addChildren(rightParenthesis);\* // adding right penth to tree as child\* nextToken(); } else { parsingError(")"); } \* // return result\* return nodeForS; } node\* A() { node\* nodeForA = new node("A");\* //creating non-terminal A\* if (currentToken().tokenInstance == "\"") { node\* quote = new node("t1", "\""); nodeForA->addChildren(quote);\* // add node for "\* nextToken();\* // move to next token\* \* // now at t2 since we had a "\* if (currentToken().tokenID == t2\_tk) { node\* tokenTwo = new node("t2", currentToken().tokenInstance); nodeForA->addChildren(tokenTwo);\* // add node for t2\* nextToken(); } else { \* // missing t2 token so error in tree\* parsingError("Needs t2\_tk for non-terminal A grammar"); } } else { node\* empty = new node("EMPTY"); nodeForA->addChildren(empty);\* // add the empty node as child\* } \* // return either (" t2) or (empty) node\* return nodeForA; } node\* B() { node\* nodeForB = new node("B");\* // creating non-terminal B node\* Token current = currentToken();\* // getting current token\* \* // adding S\* if (current.tokenInstance == "\"" || current.tokenInstance == "(") { nodeForB->addChildren(S()); } \* // adding C non-terminal\* if (current.tokenInstance == "#" || current.tokenInstance == "!") { nodeForB ->addChildren(C()); } \* // adding D non-terminal\* if (current.tokenInstance == "$") { nodeForB->addChildren(D()); } \* // adding E non-terminal\* if (current.tokenInstance == "'") { nodeForB->addChildren(E()); } \* // adding G non-terminal\* if (current.tokenID == t2\_tk) { nodeForB->addChildren(G()); } \* // returning one of the B options\* return nodeForB; } node\* C() { node\* nodeForC = new node("C");\* // creating C non-terminal node\* if (currentToken().tokenInstance == "#") { node\* hashNumber = new node("t1", "#"); nodeForC->addChildren(hashNumber); nextToken(); \* // next token is t2\* if (currentToken().tokenID == t2\_tk) { node\* t2 = new node("t2", currentToken().tokenInstance); nodeForC->addChildren(t2); nextToken(); } else { parsingError("Need t2\_tk"); } } else if (currentToken().tokenInstance == "!") { node\* exclamation = new node("t1", "!"); nodeForC->addChildren(exclamation); nextToken(); \* // Parse F\* nodeForC->addChildren(F()); } else { parsingError("Need '#' or '!' for C non-terminal grammar"); } \* // returnng one of the temrinals\* return nodeForC; } node\* D() { node\* nodeForD = new node("D");\* // creating D non-terminal node\* if (currentToken().tokenInstance == "$") { node\* dollar = new node("t1", "$");\* // creating dollar node\* nodeForD->addChildren(dollar);\* // adding dollar node\* nextToken(); \* // adding/creating F non-temrinal node\* nodeForD->addChildren(F()); } else { parsingError("Need $ for D non-terminal grammar"); } \* // returning the result\* return nodeForD; } node\* E() { node\* nodeForE = new node("E");\* // adding non-temrinal node E\* if (currentToken().tokenInstance == "'") { node\* quote = new node("t1", "'");\* // creating node for quote\* nodeForE->addChildren(quote);\* // adding quote as child for E\* nextToken(); \* // adding the F's non-terminal node\* nodeForE->addChildren(F());\* // first E\* nodeForE->addChildren(F());\* // second E\* nodeForE->addChildren(F());\* // third E\* \* // adding the B non-temrinal node\* nodeForE->addChildren(B()); } else { parsingError("Need ' for E non-terminal grammar"); } \* // return result\* return nodeForE; } node\* F() { node\* nodeForF = new node("F");\* // creating non-terminal node for F\* \* // adding t2 token\* if (currentToken().tokenID == t2\_tk) { node\* tokenTwo = new node("t2", currentToken().tokenInstance); nodeForF->addChildren(tokenTwo); nextToken(); } \* // adding t3 token\* if (currentToken().tokenID == t3\_tk) { node\* tokenThree = new node("t3", currentToken().tokenInstance); nodeForF->addChildren(tokenThree); nextToken(); } if (currentToken().tokenInstance == "&") { node\* andSymbol = new node("t1", "&"); nodeForF->addChildren(andSymbol); nextToken(); \* // now the F's\* nodeForF->addChildren(F());\* // first F\* nodeForF->addChildren(F());\* // second F\* } else { parsingError("Need t2\_tk, t3\_tk, or & for F non-termial grammar"); } \* // return result of F\* return nodeForF; } node\* G() { node\* nodeForG = new node("G"); if (currentToken().tokenID != t2\_tk) { parsingError("Need t2\_tk for G non-terminal grammar"); } node\* tokenOfTwo = new node("t2", currentToken().tokenInstance);\* // creating token 2 as node\* nodeForG->addChildren(tokenOfTwo);\* // adding token 2 as child\* nextToken(); \* // now the %\* if (currentToken().tokenInstance != "%") { parsingError("Need % for G non-terminal grammar"); } node\* percent = new node("t1", "%");\* // creating node for %\* nodeForG->addChildren(percent);\* // adding node as child to G\* nextToken(); \* // parsing F\* nodeForG->addChildren(F()); \* // return result in G\* return nodeForG; } node\* parser(const string &fileName) { \* // opening file in read\* ifstream file(fileName); tokens = scanner(file, fileName, 0); tokenSize = 0; \* // creating the root of tree (start of grammar)\* node\* root = S(); if (currentToken().tokenID != EOFTk) { parsingError("End of file"); } return root; }.

AI fix:

The issue is that you're checking for t2\_tk or t3\_tk first, and if either of them matches, you increment the token, but then you still require that the next token be '&'. This means your grammar is defined as requiring BOTH a t2/t3 token AND an '&' token, which doesn't match what you're trying to parse.

To fix this, you should use else if instead of separate if statements, or add a return after handling t2\_tk or t3\_tk cases. Here's the corrected implementation:

node\* F() { node\* nodeForF = new node("F"); // Check for t2 token if (currentToken().tokenID == t2\_tk) { node\*tokenTwo = new node("t2", currentToken().tokenInstance); nodeForF->addChildren(tokenTwo); nextToken(); returnnodeForF;  // Return after processing t2 } // Check for t3 token else if (currentToken().tokenID == t3\_tk) { node\*tokenThree = new node("t3", currentToken().tokenInstance); nodeForF->addChildren(tokenThree); nextToken(); returnnodeForF;  // Return after processing t3 } // Check for & symbol else if (currentToken().tokenInstance == "&") { node\*andSymbol = new node("t1", "&"); nodeForF->addChildren(andSymbol); nextToken(); // now the F's nodeForF->addChildren(F()); nodeForF->addChildren(F()); return nodeForF; } else { parsingError("Need t2\_tk, t3\_tk, or & for F non-termial grammar"); } return nodeForF;  // This line is technically unreachable but good practice }

*Project #*: **\_\_P2\_\_**

Grant Hughes 4/1/2025

*Signature (typed name is fine) Date*