Jacob Nguyen, James Nguyen, and Michael Lim

CPSC 323-05: Compilers and Languages

Negin Ashrafi

5/7/2025

Python Lexer Program

This program takes a text file (.txt) and converts each of the lexemes into tokens. It goes character by character to determine the lexeme state, and changes states using a state machine. The program outputs an array of arrays filled with a lexeme and its matching token. The time complexity of this program is O(n) since the program has to go through each character individually. Space complexity is O(n) since the memory usage is proportional to the input size. There were originally 2 variations for this project. The first iteration ran the same in both time and space complexity, but was very poorly constructed. We originally read the slides and looked at my notes to try to implement the program and nothing more. We were unsure of how to implement states in code, but we opted for loose logic, and simulated states. This version used implied states. The second version uses an actual state table with transitions. This more closely resembles how a real DFA works, and thus passes as a real lexer.

Code Discussion

Lines 1-28 set up the transition table to move from state to state. 31-44 has the charClassification function which categorizes each character as they are taken from the input. This is important since other tokens such as whitespace and “)” can mark the end of other lexemes. This creates the language we use to verify that each character is a valid character. 46-48 contains the “isKW” function. It is specifically for identifying keyword lexemes. We need this to be separate since we have to check to see if a lexeme is equivalent to one of our keywords after

we reach an end marker. Lines 50 - 62 contains the textImport function that is required to import text from a text file (p1Test.txt). We also add an extra \n to the end to prevent from any indexing issues in the lexer function. 67-126 contains the lexer algorithm. We take the entire string from the file we read from textImport, and create 4 variables. Lexeme holds the lexeme string that keeps track of each character that falls in line with our language rules we set up in the charClassificaition function. Tokens holds the lexeme and token pairs per valid token. State hotels the value of the state we are currently in. i is a counting variable that keeps track of each character process. Lines 85 - 87 remove comments by detecting if there is a double slash ("//"). If there are, find the next \n (indicates a new line) and go to that. This skips the rest of the line, which comments do. 89-118 is a loop that goes through each character. 89-96 deals with the start state. If the start state goes to done, we append the token we find. If the lexeme is not a whitespace or unknown, we move to the next character and change states based on the output of the charClassification function that gives us our input character's classification. 99-111 deals with the other states (IN_INT and IN_ID). The others states loop when given an digit, IN_ID loops when also given an alphanumeric character. Both stop when given any other character. If the state is done, then we process the lexeme into a token, reset the lexeme variable, and reset the state to start. We also use the isKW function to check to see if the lexeme is a keyword. If we reach an unknown character, lines 113-117 processes any lexemes we already have. 118 increments i to the next character. We also process after the while loop in case there is still a lexeme we didn't process. We then return the tokens array we made on line 126. Lines 131-137 print out the tokens.