Homework_1-Explanations

Part 1

A. Describe how you implemented the new required tokens.

For the `SEMI` token, I've added a new condition to check if the current character is a semicolon `;` and if it is, I consume the character using `self.ss.eat_char()` and return a new Lexeme with the token type `Token.SEMI` and the value `;`.

For the `INCR` token, I've modified the existing condition for the `ADD` token. After consuming the `+` character, I check if the next character is also a `+`, and if it is, then I consume the second `+` and return a new Lexeme with the token type `Token.INCR` and the value `++`. Otherwise proceed with returning the `ADD` token as before.

I allowed `ID`'s to contain numbers (as long as they're not the first character) by modifying the existing condition for `CHARS` to check if the character is also in `NUMS`.

I allowed `NUMS` to contain an optional decimal points by adding a new condition to check if the current character is a dot character `.`, and if it is, then I consume the character using `self.ss.eat_char()` and check if the next character is a number. If it's a number, then I consume it and continue consuming numbers until I encounter a dot or a non-number character. I've also handled the edge case where a number is followed by a dot and then more numbers.

B. Write down your timings for running the provided test cases. Comment on their scaling.

My time to parse for running `NaiveScanner.py` on `part1.txt` was 0.00012969970703125 seconds or 0.129699707 milliseconds.

My time to parse for running `NaiveScanner.py` on `test10.txt` was 0.00014138221740722656 seconds or 0.1413822174 milliseconds.

My time to parse for running `NaiveScanner.py` on `test100.txt` was 0.0009427070617675781 seconds or 0.9427070618 milliseconds.

In my implementation of `NaiveScanner.py`, my scanner iterates through each character in the input string once, so it performs constant-time operations for each character. Therefore, the overall time complexity is linear or O(n), where n is the length of the input string.

Part 2

A. Describe your RE definitions.

`IF`: Matches the exact string "if" using the regular expression `if`.

`ELSE`: Matches the exact string "else" using the regular expression `else`.

`WHILE`: Matches the exact string "while" using the regular expression `while`.

`INT`: Matches the exact string "int" using the regular expression `int`.

`FLOAT`: Matches the exact string "float" using the regular expression `float`.

`HNUM`: Matches a hexadecimal number starting with "0x" followed by one or more hexadecimal digits (0-9 or a-f or A-F) using the regular expression `0x[0-9a-fA-F]+`.

`ID`: Matches an identifier that starts with a letter (a-z or A-Z) and can be followed by zero or more number of letters or digits using the regular expression `[a-zA-Z][a-zA-Z0-9]*`.

NUM: Matches a number that can be either a floating-point number or an integer. It uses the regular expression `(\.[0-9]+)|([0-9]+(\.[0-9]*)?)`, which matches either a dot followed by one or more digits, or one or more digits optionally followed by a dot and zero or more digits.

`INCR`: Matches the increment operator "++" using the regular expression `\+\+`.

`PLUS`: Matches the plus sign "+" using the regular expression `\+`.

`MULT`: Matches the asterisk "*" using the regular expression `\*`.

`SEMI`: Matches the semicolon ";" using the regular expression `;`.

`LPAREN`: Matches the left parenthesis "(" using the regular expression `\(`.

`RPAREN`: Matches the right parenthesis ")" using the regular expression `\)`.

`LBRACE`: Matches the left brace "{" using the regular expression `{`.

`RBRACE`: Matches the right brace "}" using the regular expression `}`.

`ASSIGN`: Matches the equals sign "=" using the regular expression `=`.

`IGNORE`: Matches one or more whitespace characters (space, newline, or tab) using the regular expression `[ \n\t]+`.

B. Write down your timings for running the EM Scanner on the provided test cases. You do not need to run it past size 100. Comment on the performance scaling and how it relates to Part 1. Explain why the speed is faster or slower than part 1.

My time to parse for running `EMScanner.py` on `part2.txt` was 0.027344703674316406 seconds or 27.344703674 milliseconds.

My time to parse running `EMScanner.py` on `test10.txt` was 0.009692668914794922 seconds or 9.6926689148 milliseconds.

My time to parse for running `EMScanner.py` on `test100.txt` was 1.0343637466430664 seconds or 1034.3637466 milliseconds.

In my implementation of EMScanner.py, my scanner uses regular expressions to match tokens, which is computationally more expensive and thus the speed is slower than the character-by-character implementation used in NaiveScanner.py. The overall time complexity is O(n^2 * m), where n^2 is the length of the input string and nested loops, m is the number of token patterns.

Part 3

A. Describe how your SOS Scanner is different from the EM Scanner.

The difference between my SOS Scanner and EM Scanner is how they match tokens against the input string.

In my EM Scanner, each call to `token()` iterates through substrings of the input string, starting from the full length of the remaining input string and decreasing down to 1. For each substring, it tries to match all the defined token patterns using `re.fullmatch`.

In my SOS Scanner, I used `re.match` instead of `re.fullmatch` to match the token patterns only at the start of the remaining input string, without iterating through different substring lengths. My SOS Scanner iterates through the defined token patterns once per call to `token()`. It checks each pattern against the start of the remaining input string using `re.match`, and if a match is found, it consumes the matched substring from the input string and returns the corresponding lexeme. If no match is found after trying all the token patterns, my SOS Scanner raises a `ScannerException`.

My SOS Scanner is more efficient than EM Scanner because it doesn't use nested loops to check each token pattern against different substring lengths.

B. Report your timings for running the test inputs and compare against the timings for the EM Scanner (at least for 10 and 100).

My time to parse for running `SOSScanner.py` on `test10.txt` was 0.00035190582275390625 seconds (0.3519058228 milliseconds), while the time to parse for running `EMScanner.py` on `test10.txt` was 0.009692668914794922 seconds (9.6926689148 milliseconds).

My time to parse running the `SOSScanner.py` on `test100.txt` was 0.0012505054473876953 seconds (1.2505054474 milliseconds), while the time to parse for the `EMScanner.py` on `test100.txt` was 1.0343637466430664 seconds (1034.3637466 milliseconds).

Comparing these timings, my SOS Scanner is faster than the EM Scanner in both test cases.

For `test10.txt`, the SOS Scanner is approximately 27.5 times faster than the EM Scanner (0.009692668914794922 / 0.00035190582275390625).

For `test100.txt`, the SOS Scanner is approximately 827.2 times faster than the EM Scanner (1.0343637466430664 / 0.0012505054473876953).

C. Describe why there is a performance difference between the SOS and EM Scanner.

My EM Scanner uses nested loops, where it iterates through different substring lengths of the input string and tries to match each token pattern against each substring using `re.fullmatch`. This approach has a higher time complexity because it requires checking each token pattern multiple times for different substrings.

In contrast, my SOS Scanner uses `re.match` to match token patterns only at the start of the remaining input string. It iterates through the token patterns once per call to `token()` and consumes the matched substring from the input string, which is more efficient because it does not use nested loops and minimizes the number of pattern matching operations.

My SOS Scanner's time complexity is closer to O(nm), where n is the length of the input string, and m is the number of token patterns. In contrast, my EM Scanner's time complexity is closer to O((n^2)m) due to nested loops.

Part 4

A. Describe how your NG Scanner is different from the SOS Scanner.

The difference between my NG Scanner and EM Scanner is how they match tokens against the input string.

In my SOS Scanner, it iterates through the defined token patterns and tries to match each pattern against the start of the remaining input string using `re.match`. If a match is found, it consumes the matched substring and returns the corresponding lexeme.

In contrast, my NG Scanner builds one giant regular expression by combining the individual token patterns using named groups. It compiles this giant regular expression in my `build_regex` helper function. During token matching, my NG Scanner uses the compiled regular expression to match against the start of the remaining input string by using named groups to identify which token pattern matched. If a match is found, it iterates through the defined token patterns and checks which named group captured the match. It then consumes the matched substring, creates the corresponding lexeme, and returns it. My NG Scanner improves upon the SOS Scanner by reducing the number of regular expression matching operations per call to `token()`. It matches against a single giant regular expression instead of iterating through individual token patterns.

B. Report your timings for running the test inputs and compare against the timings for the SOS Scanner.

My time to parse running `NGScanner.py` on `test10.txt` was 0.00010848045349121094 seconds (0.1084804535 milliseconds), while my time to parse for the `SOSScanner.py` on `test10.txt` was 0.00035190582275390625 seconds (0.3519058228 milliseconds).

My time to parse for running `NGScanner.py` on `test100.txt` was 0.0007669925689697266 seconds (0.766992569 milliseconds), while my time to parse for `SOSScanner.py` on `test100.txt` was 0.0012505054473876953 seconds (1.2505054474 milliseconds).

Comparing these timings, my NG Scanner is faster than the SOS Scanner for both test cases.

For `test10.txt`, my NG Scanner is approximately 3.2 times faster than the SOS Scanner (0.00035190582275390625 / 0.00010848045349121094).

For `test100.txt`, my NG Scanner is approximately 1.6 times faster than the SOS Scanner (0.0012505054473876953 / 0.0007669925689697266).

C. Explain the performance difference between the SOS and NG Scanner.

In my SOS Scanner, each call to `token()` iterates through the defined token patterns and tries to match each pattern against the start of the remaining input string using `re.match`.

In my NG Scanner, it builds one giant regular expression by combining the individual token patterns using named groups, so it compiles this regular expression once during initialization.

My NG Scanner uses the compiled regular expression to match against the start of the remaining input string. My NG Scanner reduces the number of regular expression matching operations per call to `token()` by avoiding iterating through individual token patterns like in the SOS Scanner.

My NG Scanner implementation is more efficient than my SOS Scanner because it reduces the number of computationally expensive regular expression matching operations by trading off the one-time cost of building the giant regular expression for improved performance during token matching.

D. If you shuffle the tokens list in tokens.py, is the NG Scanner guaranteed to be correct? Why or why not?

If I shuffle the tokens list in `tokens.py`, the NG Scanner is not guaranteed to be correct because the order of the token patterns in the tokens list matters for the correctness of the NG Scanner. My NG Scanner builds the giant regular expression by concatenating the individual token patterns using the `|` alternation operator. The order of the patterns in the resulting regular expression depends on the order of the tokens in the tokens list. When matching against the giant regular expression, the NG Scanner will return the first matching token pattern. The tokens list should have more specific token patterns ordered before less specific token patterns. If the order of the token patterns is changed, it can lead to incorrect matches.