```
import re


class Scanner:
    def __init__(self, symbol_table):
        self.reserved_words =
re.compile(r'MORE|char|const|perform|otherwise|when|then|choose|nr|scan|write
|while|var')
        self.operators = re.compile(r'\+|-|\*|/|-
>|<|<=|>=|>|=|!=|==|:|;|%|$')
        self.separators = re.compile(r'\(|\)|<|>|{|}|\.|~| | {3}')
        self.identifiers = re.compile(r'[A-Za-z_][A-Za-z0-9_]*')
        self.digits = re.compile(r'[0-9]')
        self.chars = re.compile(r"'[A-Za-z0-9]'")
        self.strings = re.compile(r'"[^"]*"')
        self.fip_table = []
        self.current_position = -1
        self.symbol_table = symbol_table

    def add_element(self, token):
        # If the token is an identifier, get its position from the Symbol
Table
        if self.symbol_table.has(token):
            id = self.symbol_table.table.getVariableCount(token)
        else:
            id = -1  # Token is not an identifier

        self.fip_table.append((token, id))

    def buffering_from_file(self, input_file):
        with open(input_file, 'r') as file:
            source_code = file.read()

        return self.buffering(source_code)

    def buffering(self, source_code):
        i = 0

        while i < len(source_code):
            char = source_code[i]

            if char == '$':
                match = self.identifiers.match(source_code[i + 1:])
                if match:
                    token = match.group(0)
                    self.symbol_table.add(token)
                    self.add_element(token)
                    i += len(token) + 1
                else:
                    print("Lexical error: Invalid variable")
                    i += 1

            elif char.isalpha() or char == '_':
                match = self.identifiers.match(source_code[i:])
                if match:
                    token = match.group(0)
                    self.add_element(token)
```

```
                    i += len(token)
                else:
                    print("Lexical error: Invalid identifier")
                    i += 1

            elif char in "0123456789":
                match = self.digits.match(source_code[i:])
                if match:
                    token = match.group(0)
                    i += len(token)
                    while i < len(source_code) and source_code[i] != ' ' and
source_code[i] != '\n':
                        token += source_code[i]
                        i += 1
                    if token.isdigit():
                        self.symbol_table.add(token)
                        self.add_element(token)
                    else:
                        print(f"Lexical error: Invalid constant '{token}'")
                else:
                    print("Lexical error: Invalid constant")
                    i += 1

            elif char in "+-*/<=>!:;%":
                match = self.operators.match(source_code[i:])
                if match:
                    token = match.group(0)
                    self.add_element(token)
                    i += len(token)
                else:
                    print("Lexical error: Invalid operator")
                    i += 1

            elif char in "()<{}>.~":
                match = self.separators.match(source_code[i:])
                if match:
                    token = match.group(0)
                    self.add_element(token)
                    i += len(token)
                else:
                    print("Lexical error: Invalid separator")
                    i += 1

            elif char == "'":
                match = self.chars.match(source_code[i:])
                if match:
                    token = match.group(0)
                    self.add_element(token)
                    self.symbol_table.add(token)
                    i += len(token)
                else:
                    print("Lexical error: Invalid character")
                    i += 1

            elif char == '"':
                match = self.strings.match(source_code[i:])
                if match:
```

```
                    token = match.group(0)
                    self.add_element(token)
                    self.symbol_table.add(token)
                    i += len(token)
                else:
                    print("Lexical error: Invalid string")
                    i += 1

            elif char == " ":
                # Handle the "tab" as an identifier (ID)
                if source_code[i:i + 3] == "   ":
                    token = "tab"
                    self.add_element(token)
                    i += 3  # Skip the three spaces
                else:
                    i += 1  # Skip a single space

            elif char == "\n":
                i += 1  # Ignore newline

            else:
                print(f"Lexical error: Unknown character '{char}'")
                i += 1

        return self.fip_table

    def get_fip_table(self):
        return self.fip_table

    def print_tables(self):
        for code, position in self.fip_table:
            print(f"{code} {position}")

        print(self.symbol_table.__str__())

    def write_tables_to_files(self, output_fip_file,
output_symbol_table_file):
        with open(output_fip_file, 'w') as fip_output,
open(output_symbol_table_file, 'w') as symbol_table_output:
            # Write FIP table to fip_output
            for code, position in self.fip_table:
                fip_output.write(f"{code}   | {position}\n")

            symbol_table_output.write(self.symbol_table.__str__())
```