```
from LexicalError import LexicalError
from ProgramInternalFormat import ProgramInternalFormat
from SymbolTable import SymbolTable

import re


def identify_token(token):
    keywords = ["nothing", "int", "num", "text", "character", "boolean", "true", "false", "list", "dictionary",
"structure",
                "if", "else", "match", "case", "default", "break", "while", "for", "fun", "ret", "fixed", "try", "catch",
"throw", "go"]

    identifiers_regex = r"^[a-zA-Z_][a-zA-Z0-9_]*$"

    constants_regex = r"^-?(?:[0-9]+(?:\.[0-9]*)?|\.[0-9]+)(?:[eE][-+]?[0-9]+)?$"

    operators = {"+": "addition", "-": "subtraction", "*": "multiplication", "/": "division", "%": "remainder", "**":
"power", "=": "assign",
                 "+=": "i_addition", "-=": "i_subtraction", "/=": "i_division", "*=": "i_multiplication", "**=": "i_power",
"<": "less_than",
                 "<=": "less_or_equal_than", ">": "greater_than", ">=": "greater_or_equal_than", "==": "equal",
"!=": "not_equal",
                 "and": "logical_and", "&&": "logical_and", "or": "logical_or", "||": "logical_or", "++": "increment",
"--": "decrement",
                 "?": "query", "->": "return_type", ".": "selector", "//": "comment", "/*": "comment_begin", "*/":
"comment_end"}

    increment_decrement_regex = r"^(\+\+|--)?[a-zA-Z_][a-zA-Z0-9_]*(\+\+|--)?$"

    separators = {" ": "white_space", ",": "enumeration", ";": "statement_end", ":": "specifier", "&":
"reference", "(": "call_start", ")": "call_end",
                  "=>": "arrow", "[": "selector_start", "]": "selector_end", "{": "block_start", "}": "block_end"}

    if token in keywords:
        return "keyword"

    if bool(re.match(identifiers_regex, token)):
        return "identifier"

    if token in operators or bool(re.match(increment_decrement_regex, token)):
        return "operator"

    if token in separators:
        return "separator"

    if len(token) >= 2:
        if token[0] == "'" and token[len(token) - 1] == "'" or token[0] == '"' and token[len(token) - 1] == '"':
            return "string"

    if bool(re.match(constants_regex, token)):
        return "constant"

    return "error"
```

```
def test_keyword_type():
    expected_results = {'_validIdentifier': "identifier", '123Invalid': "error", '_123': "identifier", 'for': "keyword",
'if': "keyword"}

    for token, token_type in expected_results.items():
        assert identify_token(token) == token_type


def test_identifier_type():
    expected_results = {'_validIdentifier': "identifier", '123Invalid': "error", '_123': "identifier", 'valid_123':
"identifier", 'not-valid': "error"}

    for token, token_type in expected_results.items():
        assert identify_token(token) == token_type


def test_operator_type():
    expected_results = {"+": "operator", '*=': "operator", "1": "constant"}

    for token, token_type in expected_results.items():
        assert identify_token(token) == token_type


def test_separator_type():
    expected_results = {" ": "separator", '"world"': "string", "}": "separator"}

    for token, token_type in expected_results.items():
        assert identify_token(token) == token_type


def test_string_type():
    expected_results = {"'hello'": "string", '"world"': "string", "1": "constant"}

    for token, token_type in expected_results.items():
        assert identify_token(token) == token_type


def test_constant_type():
    expected_results = {'123': "constant", '-456': "constant", '3.14': "constant", '-.678': "constant", '1e10':
"constant", '123.': "constant", '.456': "constant", '-2.5E-3': "constant",
                        '"not_a_number"': "string"}

    for token, token_type in expected_results.items():
        assert identify_token(token) == token_type


def test_identify_token():
    test_keyword_type()
    test_identifier_type()
    test_operator_type()
    test_separator_type()
    test_string_type()
    test_constant_type()
```

```
def is_symbol(token):
    symbol_types = ["identifier", "constant", "string"]

    token_type = identify_token(token)

    if token_type == "error":
        raise LexicalError()

    return token_type in symbol_types


def scan(file, symbol_table, program_internal_state):
    line_number = 0
    column_number = 0

    separators_regex = r"[ ,;:&()\[\]{}]"

    with open(file) as file:
        for line in file:
            line_number += 1

            tokens = re.split(separators_regex, line.strip())
            tokens = [token for token in tokens if token]  # symbols are unique in ST

            for token in tokens:
                try:
                    if is_symbol(token) and token not in symbol_table:
                        symbol_table.add(token)

                        program_internal_state.add(identify_token(token),
position=symbol_table.position_of(token))

                    else:
                        program_internal_state.add(token)

                except LexicalError as lexical_error:
                    column_number = line.index(token) + 1

                    print(
                        "lexical error: ", f"'{token}'" + "\n" +
                        "line: ", line_number, "\n" +
                        "column: ", column_number
                    )
                    exit()


def save_to(symbol_table, output_file):
    with open(output_file, "w") as file:
        file.write("HashTable\n")

        symbols = symbol_table.symbols()
        positions = symbol_table.positions()

        for i in range(len(symbols)):
```

```
            file.write(str(positions[i]) + ": " + str(symbols[i]) + "\n")


def save_PIF_to(PIF, output_file):
    with open(output_file, "w") as file:
        tokens = PIF.tokens()
        positions = PIF.positions()

        for i in range(len(tokens)):
            file.write(str(positions[i]) + ": " + str(tokens[i]) + "\n")


def main():
    symbol_table = SymbolTable()
    program_internal_state = ProgramInternalFormat()

    # input_files = ["p1.txt", "p2.txt", "p3.txt", "error.txt"]
    input_files = ["p1.txt"]

    for file in input_files:
        print("scanning", file)

        scan(file, symbol_table, program_internal_state)

        print(file, "is", "lexically correct\n")

    save_to(symbol_table, "ST.out")
    save_PIF_to(program_internal_state, "PIF.out")


if __name__ == "__main__":
    main()
```