https://github.com/neon1024/Formal-Languages-and-Compiler-Design

Overview
This project implements a simple lexical analyzer for a custom programming language. It processes source code by reading an input file, tokenizing the text into lexical tokens, and building internal representations like a Symbol Table and Program Internal Format. The project is organized across several files:

main.py:
- token identification and scanning
- contains the main logic for identifying and classifying tokens from a source code file; it supports keywords, operators, constants, identifiers, strings, and separators.

- identify_token(token):
 - identifies and classifies a given token based on regular expressions and predefined sets of keywords, operators, and separators.
 - token types include:
  - keywords (if, else, for, etc.)
  - identifiers (valid variable names)
  - constants (numeric values, including integers and floating-point numbers)
  - operators (+, -, *, etc.)
  - separators (commas, semicolons, braces, etc.)
  - strings (enclosed in single or double quotes)
  - errors (invalid tokens)

- testing functions:
 - test_keyword_type()
 - test_identifier_type()
 - test_operator_type()
 - test_separator_type()
 - test_string_type()
 - test_constant_type()
 - is_symbol(token):
  - checks if a token is a valid symbol (i.e., an identifier, constant, or string).
  - raises a LexicalError if the token is invalid.

- scan(file, symbol_table, program_internal_state):
 - the main function for scanning an input file.
 - tokenizes each line of the file and updates the Symbol Table and Program Internal Format.
 - lexical errors are reported with line and column numbers.

- save_to(symbol_table, output_file):
 - saves the content of the Symbol Table to an output file.

- save_PIF_to(PIF, output_file):
 - Saves the Program Internal Format to an output file.

- main():
 - orchestrates the overall program, scanning input files, validating tokens, and saving the ST and PIF.

SymbolTable.py:
- manages a Symbol Table (ST) using a hash table for efficient symbol storage.

- add(symbol):
 - adds a new symbol to the table if it doesn’t already exist.

- position_of(symbol):
 - returns the position of a symbol in the table.
 - throws an exception if the symbol does not exist.

- symbols() and positions():
 - return lists of all symbols and their corresponding positions.

- clear():
 - clears the symbol table.

ProgramInternalState.py:
- manages the Program Internal Format (PIF), representing tokenized data.
- this file defines the ProgramInternalFormat class, which stores tokenized data in a hash table.

- add(token, position=-1):
 - adds a token to the internal state, along with its position (if applicable).

- tokens() and positions():
 - return lists of all tokens and their corresponding positions.

- clear():
 - clears the internal state.

HashTable.py:
- implements a custom hash table used in the ST and PIF.
- this file implements a hash table with open addressing and double hashing for collision resolution.

- __setitem__(key, value):
 - inserts or updates a (key, value) pair in the hash table using double hashing.

- __getitem__(key):
 - retrieves the value for a given key.

- __delitem__(key):
 - removes a (key, value) pair from the hash table and rehashes items to maintain consistency.

- _resize():
 - resizes the table when the load factor exceeds the threshold to reduce collisions.

- keys() and values():
 - return lists of all keys and values in the hash table.

- clear():
 - clears all elements from the hash table.

Execution Flow:
- the program starts by invoking the main() function.
- the source file(s) are read, and each line is tokenized using scan().
- tokens are classified using identify_token(), and valid tokens are added to the Symbol Table or Program Internal Format.
- lexical errors are caught and reported.
- the ST and PIF are saved to output files.