Report on Spell-Checker Project

Matheas-Roland Borsos

- **Task Overview**

The task for this project was to build a spell-checker system capable of checking whether words exist in a predefined dictionary and suggesting possible corrections for misspelled words. The system also needed to support features such as adding custom words to the dictionary, cleaning words by removing non-alphabetic characters, and splitting a text into a list of individual words.

- **Architecture of the Solution**

1. **Dictionary Representation in Dictionary.hs**: The dictionary is represented as a Map from Data.Map.Strict, where the keys are T.Text values (representing words) and the values are booleans (indicating whether the word exists in the dictionary). This design is both memory-efficient and provides fast lookups, thanks to the properties of a Map.
   Core Functions :
   - **checkWord**: This function checks whether a given word exists in the dictionary by looking it up in the Map. It uses the cleaned version of the word to ignore non-alphabetical characters.
   - **suggestCorrections**: This function suggests corrections for a given word by comparing it against words in the dictionary using the Damerau-Levenshtein distance. This allows for proximity-based suggestions, accounting for common typing errors such as swapped letters or omitted characters.
   - **addCustomWord**: This function adds a custom word to the dictionary, allowing users to extend the dictionary with their own terms.
   - **cleanWord**: This function removes any non-alphabetical characters from a word, making it easier to compare words that may contain extraneous symbols.
   - **splitWords**: This function splits a string of text into a list of words, taking care to filter out invalid characters.
2. **ErrorHandler.hs**: this file focuses on handling input/output (I/O) operations safely, ensuring robustness in the face of unexpected errors like missing files or invalid paths. Provides utility functions for safely reading from and writing to files and uses exception handling to manage I/O errors gracefully.
   Core Functions:
   - **safeReadFile:** Reads the content of a file and handles potential I/O exceptions.
   - **safeWriteFile**: Writes content to a file while ensuring proper error handling.

3. **Main.hs:** this file serves as the entry point for the project. It integrates the functionality from the other modules and provides the user-facing interface. Coordinates the flow of data between the dictionary and the user and performs I/O actions like reading dictionary files and user input.

4. **SpellChecker.hs:** handles user interaction, managing the spell-checking process using functionality from Dictionary.hs and ErrorHandler.hs. It facilitates file input/output, word checking, suggestions, and updating the dictionary interactively.
   Core Functions:
   - **run**: Main function coordinating dictionary loading, text checking, and custom word additions.
   - **localCheckWord**: Checks individual words, cleaning input and displaying errors or suggestions.
   - **addCustomWordsLoop**: Recursively prompts users to add custom words and saves updates to the dictionary.

5. **Testing in Spec.hs**: The solution includes unit tests that validate the functionality of each of these core functions. Additionally, property-like tests were created to ensure that certain general properties, such as correct handling of dictionary lookups and word cleaning, hold true across different inputs.

- **Architecture Decisions**

1. **Use of Data.Map.Strict for the Dictionary**:
   A Map from Data.Map.Strict was chosen to represent the dictionary because it provides efficient lookup operations (O(log n) complexity). Using a strict map helps to ensure that data is evaluated immediately, which is important for performance when dealing with larger datasets. This structure allows for quick access to check whether a word exists and to retrieve suggested corrections.

2. **Use of T.Text for Words**:
   T.Text from the Data.Text module was chosen for representing words instead of String because it is more memory-efficient and provides better performance, especially when working with large amounts of text. T.Text is a strict, immutable sequence of characters that is optimized for performance in Haskell.

3. **Use of Damerau-Levenshtein Distance for Correction Suggestions**:
   The Damerau-Levenshtein distance algorithm was used for suggesting corrections because it is well-suited for detecting common typographical errors, such as inserting, deleting, substituting, or transposing characters. This choice allows the spell checker to generate sensible and relevant suggestions when a user misspells a word.

4. **Efficient Word Cleaning**:
   The cleanWord function removes any non-alphabetical characters to standardize the words before checking them against the dictionary. This approach ensures that the spell-checker can handle dirty inputs, such as words with punctuation marks, while minimizing errors due to character mismatches.

5. **Separation of Concerns**:
   The architecture separates the core spell-checking functionality from the I/O operations, such as reading from and writing to files. This makes it easy to test the core functions independently and keep the logic decoupled from external dependencies.

- **Library Choices**

1. **Data.Map.Strict**:
   Chosen for its efficient implementation of a key-value store. It provides fast lookups and is suitable for handling the dictionary, ensuring efficient performance as the dictionary grows.

2. **Data.Text**:
   T.Text was selected for representing strings to achieve better performance in terms of memory usage and speed. It is more efficient than String for handling text, especially in large-scale applications.

3. **Test.Hspec**:
   This library was used for writing unit tests and property-like tests. Hspec is a popular testing framework in the Haskell community that allows for expressive and easy-to-read test definitions.

4. **Control.Exception**:
   Used for handling exceptions, specifically for I/O operations such as reading from and writing to files. This library ensures that the system can handle any potential runtime errors, such as missing files or invalid file paths.

5. **sortOn**:
   Used for sorting the list of suggested corrections based on their Damerau-Levenshtein distance. This function helps to ensure that the most relevant suggestions appear first in the list.

- **Investigation of Performance**

Performance was a key consideration throughout the development of the spell-checker system. The major factors influencing performance include the efficiency of the dictionary lookup, the Damerau-Levenshtein distance calculation, and the text processing (such as cleaning and splitting words).

1. **Dictionary Lookups**:
   Using Data.Map.Strict for storing the dictionary ensures that lookups are performed in O(log n) time, which is efficient for large datasets. This provides a good balance between time complexity and memory usage.

2. **Damerau-Levenshtein Distance**:
   Calculating the Damerau-Levenshtein distance between the misspelled word and each word in the dictionary can be computationally expensive, especially as the dictionary grows. The algorithm is executed only on words that are likely to be close matches.

3. **Word Cleaning and Splitting**:
   The cleanWord function is efficient because it only performs a single pass over the word to remove non-alphabetical characters. The splitWords function processes the input string once and filters out invalid characters, making it fast enough for typical inputs.