Github Link: https://github.com/DenisaToarcas/UBB--Computer-Science/tree/main/3rd%20year/Formal%20Languages%20and%20Compiler%20Design/Lab%202

Requirements:

The assignment involves creating and managing Symbol Tables with the following specifications:

1. **Symbol Table**:
    - Two separate tables: one for identifiers and one for constants. You are required to create two instances of symbol tables.
2. **Data Structure**:
    - The Symbol Table will be implemented using a **Hash Table** data structure.

Implementation Overview:

The **approach** for my HashTable<K, V> is to use an array of lists (separate chaining) to handle collisions. Each bucket in the hash table contains a list of key-value pairs, allowing multiple entries with the same hash value.

HashTable<K, V>:

- **Purpose**: This class is a generic implementation of a hash table where K is the type of the key, and V is the type of the value. The table stores key-value pairs using generics, allowing for different types of values to be stored.
- **Collision Handling**: Separate chaining is used to resolve collisions by allowing each bucket in the table to hold a list of pairs.
- **Use Case**: The symbol tables for identifiers will store String to String pairs, while the symbol table for constants will store String to Object pairs (to accommodate both integer and string constants).

Design Choices:

- I chose this implementation to accommodate the requirement of having two separate symbol tables:
    - One for **Identifiers**, which will map strings to their types (e.g., x -> int, y -> float).
    - One for **Constants**, which will map strings to various constant values (e.g., PI -> 3.14159, MAX_INT -> 2147483647).

Key Elements:

- **Parameters**:
    - size (integer): Defines the size of the hash table (number of buckets).
    - K: The generic type of the key.

- V: The generic type of the value.

Methods:

1. **getSize()**:
    - Returns the size of the hash table (number of buckets).
2. **hash(K key)**:
    - Computes the hash code for the given key.
    - The hash code is calculated using the hashCode() of the key and then applying a modulo operation with the size of the table. This determines the bucket index where the pair will be stored.
3. **findPositionOfKey(K key)**:
    - Finds the position of a key in the hash table. It returns the bucket index and the index within that bucket where the key is found (or null if not found).
4. **findByPosition(Pair<Integer, Integer>):**
    - Finds the key from a given position in the hashtable. It returns the key or throws an "Invalid position" error if the position given is out of bounds.
5. **containsKey(K key)**:
    - Checks whether a key already exists in the hash table.
6. **add(K key, V value)**:
    - Adds a key-value pair to the hash table. It computes the hash of the key and places the pair in the appropriate bucket. If the key already exists, the method returns false and does not add the new value.
7. **findValueByKey(K key)**:
    - Retrieves the value associated with a given key.
8. **toString()**:
    - Returns a string representation of the hash table, showing the contents of all the buckets and their key-value pairs.

Pair<K, V>:

- **Purpose**: This class represents a key-value pair used by the HashTable class to store entries in its buckets.

Methods:

1. **getKey()**:

- Returns the key stored in the pair.

2. **getValue()**:
    - Returns the value stored in the pair.
3. **toString()**:
    - Provides a string representation of the key-value pair in the format (key -> value).

This implementation fulfills the requirements of creating a symbol table for both **identifiers** and **constants** using a hash table with **separate chaining** for collision resolution. The use of generics makes it flexible to store various data types, allowing the constants symbol table to hold both numeric and string values.

SymbolTable Class:

This class serves as a **wrapper** around two separate hash tables for managing identifiers and constants in a symbol table system. It encapsulates two distinct symbol tables: one for identifiers and one for constants.

- The SymbolTable class handles two distinct symbol tables:
    1. **Identifiers Symbol Table** (identifiersSymbolTable): Stores key-value pairs where both the key and value are strings (variable names and their types).
    2. **Constants Symbol Table** (constantsSymbolTable): Stores key-value pairs where the key is a string and the value can be either a string or an int (i.e., Object type).

    This class facilitates the following operations: adding, checking for the existence of keys, retrieving values by keys, and finding the position of a key for both identifiers and constants.
- **HashTable Usage**:
    - The class utilizes two instances of the generic HashTable:
        - HashTable<String, String>: For identifiers, mapping variable names to their types.
        - HashTable<String, Object>: For constants, mapping names to values (which can be either numbers or strings).

Class Components:

- size (Integer): The size of the symbol table, representing the number of buckets in the underlying hash tables.
- identifiersSymbolTable (HashTable<String, String>): Hash table for storing identifiers.

- constantsSymbolTable (HashTable<String, Object>): Hash table for storing constants.

Key Methods:

1. **Getters and Setters**:
    - **getSize() / setSize()**: Methods to retrieve and modify the size of the symbol table.
    - **getIdentifiersSymbolTable() / setIdentifiersSymbolTable()**: Get or set the hash table that stores the identifiers.
    - **getConstantsSymbolTable() / setConstantsSymbolTable()**: Get or set the hash table that stores the constants.
2. **Find Operations**:
    - **findByPositionInIdentifiersST(Pair<Integer, Integer> position)**: Finds the value (identifier) in the identifier symbol table at a specified position.
    - **findByPositionInConstantsST(Pair<Integer, Integer> position)**: Finds the value (constant) in the constants symbol table at a specified position.
    - **findValueByKeyInIdentifiersST(String key)**: Retrieves the value associated with a specific key in the identifiers symbol table.
    - **findValueByKeyInConstantsST(String key)**: Retrieves the value associated with a specific key in the constants symbol table.
3. **Containment Check**:
    - **containsKeyInIdentifiersST(String symbol)**: Checks if the key (identifier) exists in the identifier symbol table.
    - **containsKeyInConstantsST(String symbol)**: Checks if the key (constant) exists in the constants symbol table.
4. **Add Operations**:
    - **addInIdentifiersST(String symbol, String value)**: Adds a new identifier and its value (type) to the identifier symbol table. Returns true if the identifier was added successfully, false if it already exists.
    - **addInConstantsST(String symbol, Object value)**: Adds a new constant and its value to the constants symbol table. Returns true if the constant was added successfully, false if it already exists.
5. **Position Finding**:
    - **findPositionOfKeyInIdentifiersST(String symbol)**: Returns the position (bucket index and position within the bucket) of the identifier in the identifiers symbol table.

- **findPositionOfKeyInConstantsST(String symbol)**: Returns the position (bucket index and position within the bucket) of the constant in the constants symbol table.

6. **toString()**:
    - **toString()**: Returns a string representation of the symbol table by combining the string representations of both the identifiers and constants tables.