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

Symbol Table

a. unique for identifiers and constants (create one instance of ST)

The symbol table was implemented using a Hash Table in the form of a list of lists.

```
def __init__(self, capacity):
    self.capacity = capacity
    self.hashtable = [[]] * capacity
```

At a given position determined by the hash function (represented below), the list will contain all the identifiers/constants that have the same value within the hash function.

```
def hashFunction(self, k: str):
    return sum(map(ord, k)) % self.capacity
```

The hash function will compute the sum of ASCII characters modulo the capacity (so it stays within the range of the hash table). The hash function isn't perfect, since an example such as : "ab" -> 97 + 98 = 185

"ba" -> 98 + 97 = 185

would represent a collision, but it is good enough to provide a quite decent lookup time.

Lookup(search):

```
def get(self, value):
    pos = self.hashFunction(value)
    if value not in self.hashtable[pos]:
        return -1
    return pos
```

We compute the position with the hash function. If the value is not in the list at the "pos" position, it will return -1, otherwise, it will return the position.

Insert:

```
def insert(self, value: str):
    pos = self.get(value)
    if pos == -1:
        hashkey = self.hashFunction(value)
        self.hashtable[hashkey].append(value)
        return hashkey

    return pos
```

First, we try to get the position of the value in the hash table. If the value cannot be found, we insert it at the position computed by the hash function and return the position. Otherwise, we return -1.

By using a hash table as the representation of the symbol table, we get the following complexities :

- Lookup :
  - Average : O(1)
  - Worst case : O(n)
- Insert :
  - Average : O(1)
  - Worst case : O(n)
  - The insert complexity is influenced mostly by the lookup operation