Documentatie Lab3 Week 3 LFTC

Socaciu Mihai 938

**Github link:** <https://github.com/Socca98/LFTC-2020>

Lab3

Important code snippets

is\_identifier(word):

return re.match(r'**^[a-zA-Z]([a-zA-Z]|[0-9]){0,8}$**', word) is not None

is\_constant(token):

return re.match(r'**((\'[a-zA-Z]\'|\'[0-9]\')|(([+\-])?[1-9]\d\*|0)$)**', token) is not None

def tokenize(self):  
 *"""  
 Read from the file. Create regex for separators. Tokenize each*

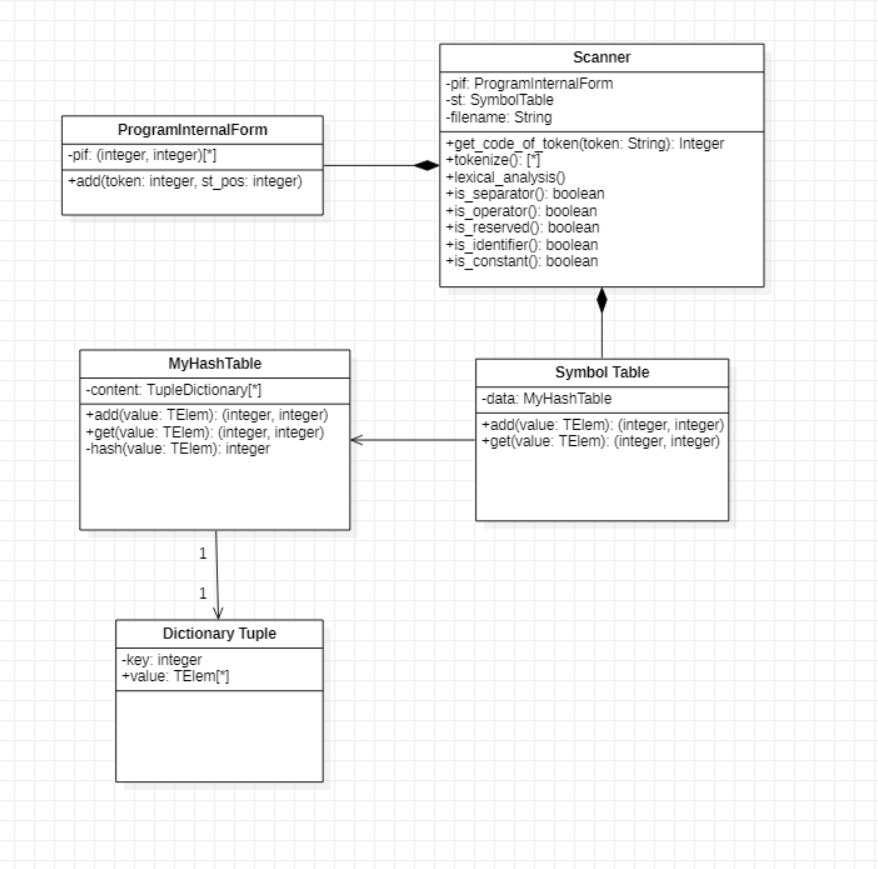
*line.*

*Each list inside the big result list is a code line of our program.* ***:return****: List of lists. Tokens from each line  
 """*

def lexical\_analysis(self):  
 *"""  
 Main algorithm for returning the completed PIF table along with the Symbol Table.  
 After tokenization of the txt file, we parse each token and check*

*if its a constant, reserved keyword, etc. Then add it to* ***PIF****.  
 If identifier/constant, we add in* ***Symbol Table*** *a record of the new*

*position in its Hash Table.  
 """*



Lab2

Domain

Ht = {ht | ht is a hash table with elements of the type TElem}

Ex: {65: ['A'], 15: ['AT', 'TA']}

Interface for Hash Table

* init(ht)
  + **Description**: creates a new empty hash table (dictionary)
  + **Pre**: True
  + **Post**: ht Ht
* add(ht, e)
  + **Description**: hashes an element e for a key and links it to that key in the ht
  + **Pre**: ht Ht, e is a TElem
  + **Post**: ht’ Ht, ht’ = ht e
* get(ht, e)
  + **Description**: Returns the position of an element if exists. Position consists of index\_key in the dictionary and index in the list linked to that position.
  + **Pre**: ht Ht
  + **Post**: get <- (index\_key, cl\_index), e is a TElem

index\_key is the key integer after hashing

cl\_index is the index in the list linked on index\_key

* hash(ht, e)
  + **Description**: Hashes an element. Sum of ASCII characters % 67.
  + **Pre**: ht Ht, e is a TElem
  + **Post**: hash <- hash\_value

Representation

content: Dictionary{}

Dictionary

key: Integer

collision\_list: TElem[]

//Hashing function algorithm

**subalgorithm** hash(ht, e) **is**:

sums\_chars <- 0

**for** i <- 0, e.length **execute**

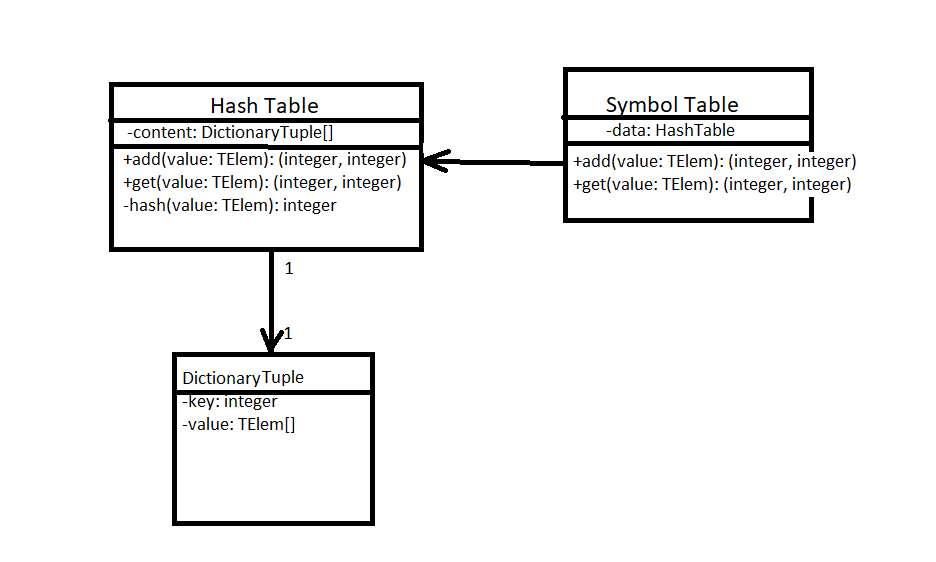
sums\_chars <- sums\_chars + (ASCII of e[i])

**end-for**

hash <- sums\_chars % 67

**end-subalgorithm**

Collision resolution

I chose **separate chaining**. Every key in the hash table has a list of elements where unique new elements with the same key are appended.