**FLCD Lab. ‘Lexical Analyser’**

- documentation –

**GitHub Repository**:

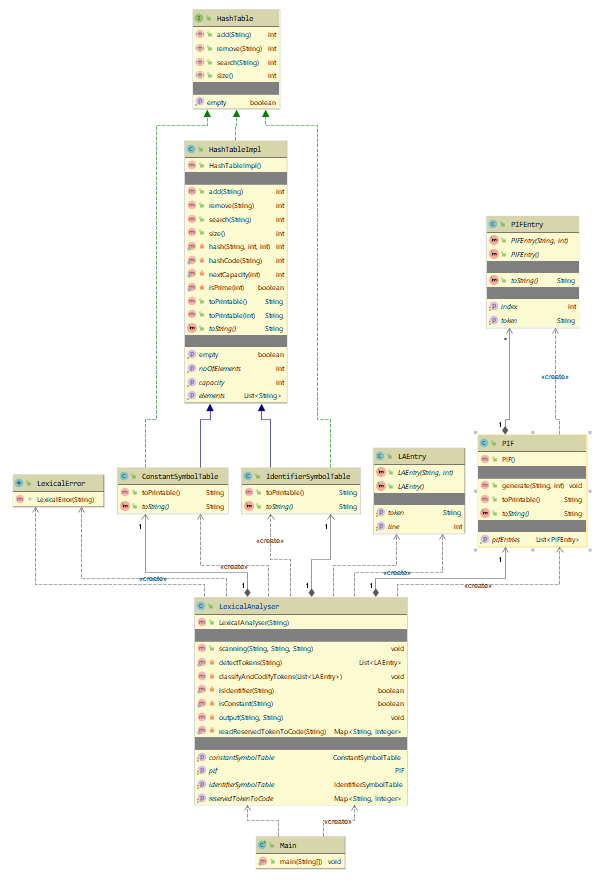
<https://github.com/adrianPascan/FLCD_Lab2>

**DATA STRUCTURE used for Symbol Table**

* *HASH TABLE*
  + m: capacity, prime number
  + elems: array list of ‘elem’
  + elem: element, string
* *hash function*: double hashing
  + c: hash code of ‘elem’
  + i: probe index,
  + *1st hash function*
  + *2nd hash function*
  + *main hash function*
* collision resolution
  + for every ‘elem’ having a specific hash code, the hash function generates a probe sequence of hash values which is a permutation of the set {0, .., m-1} given the two hash sub-functions
  + if the ‘elems’ is full, i.e. the probe sequence of ‘elem’ is already hashed to other ‘elem’s, then a resize and a rehash operation will be performed; the new capacity m’ will be equal to the first prime number p with p >= 2\*m+1

**CLASS SPECIFICATION**

**Class Diagram:**

****

**HashTable interface**

1. int *add*(String element)
   * Adds a new element to the hash table.
   * preconditions: ‘element’ is a string
   * postconditions: ‘element’ was added
   * return: the hash code of ‘element’
2. int *remove*(String element)
   * Removes an element from the hash table if it exists in the hash table.
   * preconditions: ‘element’ is a string
   * postconditions: ‘element’ was removed if it existed
   * return: the hash code of ‘element’, if it was found; -1, otherwise
3. int *search*(String element)
   * Searches for an element in the hash table.
   * preconditions: ‘element’ is a string
   * postconditions: the returned value indicates the actual existence of ‘element’ in the hash table
   * return: the hash code of ‘element’, if it was found; -1, otherwise
4. int *size*()
   * Returns the size of the hash table.
   * preconditions: True
   * postconditions: the returned size is the same as the actual size
   * return: the size of the hash table
5. boolean *isEmpty*()
   * Checks if the hash table is empty.
   * preconditions: True
   * postconditions: the returned value indicates if the hash table is actually empty
   * return: True, if the hash table is empty; False, otherwise

**HashTableImpl class**

* implements the *HashTable* interface
* fields
  1. *capacity*: int, private
  2. *noOfElements*: int, private
  3. *elements*: list of string, private
* other methods and functions
  1. private static int *hash*(String element, int capacity, int probeIndex)
     + Returns the hash value of order ‘probeIndex’ of the element ‘element’ for a hash table with capacity ‘capacity’.
     + preconditions: ‘element’ is a string; ‘capacity’, ‘probeIndex’ are positive integers
     + postconditions: the returned hash value is the same as the actual hash value
     + return: the hash value
  2. private static int *hashCode*(String element)
     + Returns the hash code of a given ‘element’.
     + preconditions: True
     + postconditions: the returned hash code is the same as the actual hash code
     + return: the hash code
  3. private static int *nextCapacity*(int capacity)
     + Returns the capacity of a hash table following capacity ‘capacity’.
     + preconditions: ‘capacity’ is an integer
     + postconditions: the returned capacity is the same as the capacity following ‘capacity’
     + return: the next capacity
  4. private static boolean *isPrime*(int no)
     + Checks if a number ‘no’ is prime.
     + preconditions: ‘no’ is an integer
     + postconditions: the returned value indicates the primness of ‘no’
     + return: True, if ‘no’ is prime; False, otherwise
  5. public String *toPrintable*()
     + Returns a printable string representation of the HashTableImpl class instance.
     + preconditions: True
     + postconditions: returned string represents a printable string representation
     + return: string
  6. public String *toPrintable*(int tabNo)
     + Returns a printable string representation of the HashTableImpl class instance indexed by ‘tabNo’ tabs.
     + preconditions: tabNo is an integer
     + postconditions: returned string represents a printable string representation indexed by ‘tabNo’ tabs
     + return: string

**ConstantSymbolTable class**

* class implementing the ST for constants
* implements *HashTable* interface
* extends *HashTableImpl* class
* other methods
  1. public String *toPrintable*()
     + Returns a printable string representation of the ConstantSymbolTable class instance.
     + preconditions: True
     + postconditions: returned string represents a printable string representation
     + return: string

**IdentifierSymbolTable class**

* class implementing the ST for identifiers
* implements *HashTable* interface
* extends *HashTableImpl* class
* other methods
  1. public String *toPrintable*()
     + Returns a printable string representation of the IdentifierSymbolTable class instance.
     + preconditions: True
     + postconditions: returned string represents a printable string representation
     + return: string

**PIFEntry class**

* auxiliary class for PIF class
* fields
  + *token*: string, private
  + *index*: int, private

**PIF class**

* class implementing the Program Internal Form (PIF)
* fields
  + *pifEntries*: list of PIFEntry, private
* methods and functions

1. public void *generate*(String token, int index)
   * Generates a PIFEntry instance ‘pifEntry’ and adds it to ‘pifEntries’.
   * preconditions: ‘token’ is a string; ‘index’ is an integer
   * postconditions: ‘pifEntry’ was added
2. public String *toPrintable*()
   * Returns a printable string representation of the PIF class instance.
   * preconditions: True
   * postconditions: returned string represents a printable string representation
   * return: string

**LAEntry class**

* auxiliary class for LexicalAnalyser class
* fields
  + *token*: string, private
  + *line*: int, private

**LexicalError class**

* extends RuntimeException class

**LexicalAnalyser class**

* class implementing the Scanner/Lexical Analyser
* fields
  + *pif*: PIF instance
  + *identifierSymbolTable*: IdentifierSymbolTable instance
  + *constantSymbolTable*: ConstantSymbolTable instance
  + *reservedTokenToCode*: Map<String, Integer>
* methods and functions
  1. public void *scanning*(String fileName, String programDirectoryPath, String outputDirectoryPath)
     + Performs a scan/lexical analysis on the source code read from file.
     + preconditions: ‘fileName’ is a valid file name inside a valid directory path ‘programDirectoryPath’; ‘outputDirectoryPath’ is a valid directory path
     + postconditions: the source code was correctly scanned
  2. private static List<LAEntry> *detectTokens*(String programFilePath)
     + Performs detection of tokens on the file and returns a list of LAEntry instances accordingly.
     + preconditions: ‘programFilePath’ is a valid file path
     + postconditions: the items of the list were correctly detected
     + return: the list
  3. private void *classifyAndCodifyTokens*(List<LAEntry> laEntries) throws LexicalError
     + Performs classification and codification on the given list of LAEntry instances.
     + preconditions: laEntries is a list of LAEntry instances
     + postconditions: the items of the list were correctly classified and codified
     + error: LexicalError, if at least one token cannot be classified
  4. private static boolean *isIdentifier*(String token)
     + Checks if a given token is an identifier.
     + preconditions: token is a string
     + postconditions: the returned value indicates if the token is an identifier
     + return: True, if ‘token’ is an identifier; False, otherwise
  5. private static boolean *isConstant*(String token)
     + Checks if a given token is a constant.
     + preconditions: token is a string
     + postconditions: the returned value indicates if the token is a constant
     + return: True, if ‘token’ is a constant; False, otherwise
  6. private void *output*(String fileName, String outputDirectoryPath)
     + Outputs the PIF and Symbol Tables of a given source code file to a given directory.
     + preconditions: ‘fileName’ is a valid file name inside a valid directory path ‘outputDirectoryPath’
     + postconditions: the parameters were outputted accordingly
  7. private static Map<String, Integer> *readReservedTokenToCode*(String reservedTokenFilePath)
     + Returns a dictionary mapping the reserved tokens to their codes read from the file.
     + preconditions: ‘reservedTokenFilePath’ is a valid file path
     + postconditions: returned map contain the tuples read from the file
     + return: the dictionary

**RUN EXAMPLES**

Token.in

main,2  
return,3  
read,4  
write,5  
true,6  
false,7  
equals,10  
lessEqual,11  
lessThan,12  
greaterEqual,13  
greaterThan,14  
if,20  
else,21  
for,22  
int,30  
string,31  
;,40  
{,41  
},42  
(,43  
),44  
[,45  
],46  
+,50  
-,51  
\*,52  
/,53  
%,54  
=,55

1. lexically CORRECT source code

Input

int main ( )  
{  
 int no = 0 ;  
 int no2 = 0 ;  
 int no3 = 0 ;  
 int minimum = 0 ;  
 read no ;  
 read no2 ;  
 read no3 ;  
 minimum = no ;  
 if ( minimum lessThan no2 ) {  
 minimum = no2 ;  
 }  
 if ( minimum lessThan no3 ) {  
 minimum = no3 ;  
 }  
 write minimum ;  
 return 0 ;  
}

Output

PIF:   
[  
 int -> -1  
 main -> -1  
 ( -> -1  
 ) -> -1  
 { -> -1  
 int -> -1  
 no -> 3521  
 = -> -1  
 0 -> 48  
 ; -> -1  
 int -> -1  
 no2 -> 109201  
 = -> -1  
 0 -> 48  
 ; -> -1  
 int -> -1  
 no3 -> 109202  
 = -> -1  
 0 -> 48  
 ; -> -1  
 int -> -1  
 minimum -> 1064538126  
 = -> -1  
 0 -> 48  
 ; -> -1  
 read -> -1  
 no -> 3521  
 ; -> -1  
 read -> -1  
 no2 -> 109201  
 ; -> -1  
 read -> -1  
 no3 -> 109202  
 ; -> -1  
 minimum -> 1064538126  
 = -> -1  
 no -> 3521  
 ; -> -1  
 if -> -1  
 ( -> -1  
 minimum -> 1064538126  
 lessThan -> -1  
 no2 -> 109201  
 ) -> -1  
 { -> -1  
 minimum -> 1064538126  
 = -> -1  
 no2 -> 109201  
 ; -> -1  
 } -> -1  
 if -> -1  
 ( -> -1  
 minimum -> 1064538126  
 lessThan -> -1  
 no3 -> 109202  
 ) -> -1  
 { -> -1  
 minimum -> 1064538126  
 = -> -1  
 no3 -> 109202  
 ; -> -1  
 } -> -1  
 write -> -1  
 minimum -> 1064538126  
 ; -> -1  
 return -> -1  
 0 -> 48  
 ; -> -1  
 } -> -1  
]

Identifier ST:   
[  
 Hash Table:   
 [  
 capacity= 11  
 noOfElements= 4  
 elements=[null, no, null, minimum, no2, no3, null, null, null, null, null]  
 ]  
]  
  
Constant ST:   
[  
 Hash Table:   
 [  
 capacity= 11  
 noOfElements= 1  
 elements=[null, null, null, null, 0, null, null, null, null, null, null]  
 ]  
]

1. lexically INCORRECT source code

Input

int main ( )  
{  
 int no = ~ ;  
 int no2 = 0 ;  
 int no3 = 0 ;  
 int 0minimum = 0 ;  
 read no ;  
 read no2 ;  
 read no3 ;  
 0minimum = no ;  
 if ( 0minimum lessThan no2 ) {  
 0minimum = no2 ;  
 }  
 if ( 0minimum lessThan no3 ) {  
 0minimum = no3 ;  
 }  
 write 0minimum ;  
 return 0 ;  
}

Output

p1err.txt-> LEXICAL ERROR: token '~' cannot be classified (line 3)