https://github.com/ElinaBarabas/FLCD

HashTable.py

For the constants and identifiers we will use two different ST, as specified in the assignment.

We will have the same hash function (sum of ascii codes % the size of the hash table), because constants can be

strings too. In this case, it is easier to build two separate hash tables, based on the same idea.

For each key in the hash table, we will have a unique value, which will act as an identifier for the given

constant/identifier.

create_entries(self) - we will create the hash table with m - "None" values for the initialization part, where m

is the size of the hash table.

hash_function(self, key) - the key will represent in our case an identifier or a constant; for this key, we will compute

the sum of the ascii codes (both for strings/integers) and for it, we will have the sum modulo m as the returned result;

add_key(self, key) - firstly, we need to check if the given key as parameter is already in the ST;

if present, we don't have to add it again; otherwise, we need to compute where to place it in the hash table;

for adding, we use a tuple in which we will find the given key with a unique value associated to it.

search_key(self, key) - this function returns a boolean by computing the hash function for the given parameter; after, it checks if the

current result is None (which means that the key has not been added yet in the hash table) or not (the key is already in the hash table);

get_hashtable(self) - it returns the hash table itself

populate_token_list(self) - based on the "token.txt", we will extract in a list, characters and keywords that must not be taken into account

as constant/identifier

count_tokens(self) - this function returns the number of tokens from "token.txt"

check_if_const_integer(self, elem) - this function checks if the current elem is associated as an integer or as string to an identifier

print_hash_tables(self) - this function lists the hash tables for the constants/identifiers of the current file (problem)

retrieve_value(self,key) - this function returns the unique value associated to the key (constant/identifier) from the Symbol Table

search_key(self,key) - this function checks if a key appears in the ST or not

get_hashtable(self) - it returns the ST

SymbolTable.py

identify_tokens(self) - it reads from token.in and creates an entry with an unique value associated for each available token; besides that, it sets value "0" for every constant and "1"

for every identifier that will appear in the Program Internal Form

count_tokens(self) - it returns the number of available tokens (from token.in)

check_if_const_integer(elem) - we need to distinguish the integer constants from the string ones; this function checks if the given element is an integer or not

identify_line_components(self) - this function splits the given program based on spaces, and it creates entries for each element on the line in the pif table

append_without_new_line(self, elem) - it identifies the unnecessary "\n" and it appends the last elements from the program's lines without it

create_hashtable_identifiers(self, components_of_line) - it decides how line's components should be grouped and in which Symbol Table they should be added

write_output_in_files(self) - based on the input problem, it creates output files which contains the STs, PIF and eventually lexical errors

create_pif_table(self, components_of_line) - it adds, in order, each line's component with an associated id, based on component's type (token/identifier/constant)

check_elem(self, elem) - for each elem, it checks if it is correctly, from a lexical point of view; if it is not correct, it retrieves the line number where the error occurs

FiniteAutomata.py

processFile(self): it analyzes the given file and it sets the set of states/transitions/q0/q

checkDeterministicFinal(self): it checkes that for each pair of source & nonterminal there is only one associated value

checkSequence(self, inputCharacterSequence): this function checks if the given sequence has a fluent transition from the initial state to the final one

The FA part is implemented in the scanner, where the constant/identifier is about to be inserted in the ST. If the check function is returning false, it means that we have an error and a message is printed, with the line number of it.

EBNF FA

STATES=STATE { " " STATE}

STATE = LETTER

DIGIT="0" | "1" | ... | "9"

ALPHABET = ELEMENT {" " ELEMENT}

ELEMENT = LETTER | DIGIT

LETTER = "a" | "b" | ... | "z" | "A" | ... | "Z"

INITIAL_STATE = STATE

FINAL_STATES = STATE {" " STATE}

TRANSITIONS = TRANSITION {"\n" TRANSITION}

TRANSITION= (STATE "," ELEMENT) " : " STATE

FA= STATES "\n" ALPHABET "\n" INITIAL_STATE "\n" FINAL_STATES "\n" TRANSITIONS