https://github.com/917-SzaboBalazs/FLCD/tree/main/lab3

Scanner Class

The `Scanner` class is responsible for scanning a program, tokenizing it, and generating the PIF (Program Internal Form) and Symbol Table.

Attributes

- `program_name` (str): The name of the input program.
- `operators` (list): A list of operators and separators in the program.
- `reserved_words` (list): A list of reserved words in the program.
- `identifier_regex` (str): Regular expression for matching identifiers.
- `constant_regex` (str): Regular expression for matching constants.
- `symbolTable` (instance of SymbolTable): An instance of the SymbolTable class to manage symbol table entries.
- `pif` (instance of Pif): An instance of the Pif class to manage the Program Internal Form.
- `tokenizer` (instance of Tokenizer): An instance of the Tokenizer class for tokenizing the input program.

Methods

`__init__(self, program_name)`

Constructor for the Scanner class.

- Parameters:
  - `program_name` (str): The name of the input program.

`_is_operator_or_separator(self, token)`

Check if a token is an operator or separator.

- Parameters:
  - `token` (str): The token to check.

`_is_keyword(self, token)`

Check if a token is a reserved keyword.

- Parameters:
  - `token` (str): The token to check.

`_is_identifier(self, token)`

Check if a token is an identifier.

- Parameters:
  - `token` (str): The token to check.

`_is_constant(self, token)`

Check if a token is a constant.

- Parameters:
  - `token` (str): The token to check.

`scan(self)`

Scan the input program, tokenize it, and generate the PIF and Symbol Table. Prints any lexical errors found.

`log_to_file(self)`

Write the PIF and Symbol Table to files ("pif.out" and "st.out").

Pif Class

The `Pif` class represents the Program Internal Form, which is used to store tokens and their corresponding positions in the Symbol Table.

Attributes

- `table` (list): A list to store tuples of tokens and their positions in the Symbol Table.

Methods

`__init__(self)`

Constructor for the Pif class.

`add(self, token, pos)`

Add a token and its position in the Symbol Table to the PIF.

- Parameters:
  - `token` (str): The token to add.
  - `pos` (int): The position of the token in the Symbol Table.

`size(self)`

Get the size of the PIF.

`get_item(self, index)`

Get the item at a specific index in the PIF.

`get_all(self)`

Get all items in the PIF.

SymbolTable Class

The `SymbolTable` class is responsible for managing the symbol table, which stores identifiers and their positions.

Attributes

- `table` (list): A list to store linked lists of symbol table entries.
- `size` (int): The number of entries in the symbol table.
- `capacity` (int): The capacity of the symbol table.

Methods

`__init__(self, capacity=100)`

Constructor for the SymbolTable class.

- Parameters:
  - `capacity` (int): The initial capacity of the symbol table.

`insert(self, key, value)`

Insert a symbol table entry.

- Parameters:
  - `key` (int): The key (position) of the entry.
  - `value` (str): The value (identifier) of the entry.

`find_by_value(self, value)`

Find a symbol table entry by its value (identifier).

- Parameters:
  - `value` (str): The value to search for.

`find(self, key)`

Find a symbol table entry by its key (position).

- Parameters:
  - `key` (int): The key to search for.

`remove(self, key)`

Remove a symbol table entry by its key (position).

- Parameters:
  - `key` (int): The key to remove.

`get_all(self)`

Get all symbol table entries.

`size(self)`

Get the current size of the symbol table.

`capacity(self)`

Get the capacity of the symbol table.

Tokenizer Class

The `Tokenizer` class is responsible for tokenizing the input program.

Attributes

- `lines` (list): A list of program lines.
- `split_symbols` (list): A list of split symbols (operators and separators).
- `program_name` (str): The name of the input program file.

Methods

`__init__(self, split_symbols, program_name=None)`

Constructor for the Tokenizer class.

- Parameters:
  - `split_symbols` (list): A list of split symbols.
  - `program_name` (str, optional): The name of the input program file.

`read_program(self, program_name)`

Read and store the lines of the input program.

- Parameters:
  - `program_name` (str): The name of the input program file.

`_strip_newlines(self)`

Remove empty rows from the list of program lines.

`_remove_whitespaces(self)`

Remove whitespaces and comments from the program lines.

`_tokenize(self)`

Tokenize the program lines using regular expressions and split symbols.

`get_tokens(self)`

Tokenize the input program and return a list of tokens.

Finite Automaton (FA) Class Documentation

The `FA` class is designed to represent a Finite Automaton, a mathematical model of computation. This class encapsulates the properties and behavior of a finite automaton, providing methods to interact with its definition and determine if a given sequence is accepted by the automaton.

Constructor

`__init__(self)`

The constructor initializes the `FA` object with the following instance variables:

- `__input_file`: The path to the input file.
- `__all_states`: A list containing all states of the automaton.
- `__input_symbols`: A list containing all input symbols.
- `__initial_state`: The initial state of the automaton.
- `__final_states`: A list containing all final states of the automaton.
- `__transition_function`: A dictionary representing the transition function of the automaton.

Methods

`get_input_file(self)`

Returns the path of the input file.

`get_all_states(self)`

Returns a list of all states in the automaton.

`get_input_symbols(self)`

Returns a list of input symbols.

`get_initial_state(self)`

Returns the initial state of the automaton.

`get_final_states(self)`

Returns a list of final states of the automaton.

`get_transition_function(self)`

Returns the transition function of the automaton as a dictionary.

`read_from_file(self, input_file)`

Reads the automaton definition from a specified input file. It populates the instance variables based on the content of the file.

- Parameters:
  - `input_file`: Path to the input file.

`seq_is_accepted(self, seq)`

Checks if a given sequence is accepted by the automaton.

- Parameters:
  - `seq`: The input sequence to be checked.
- Returns:
  - `True` if the sequence is accepted, `False` otherwise.

Example Usage

```python
# Instantiate FA object
fa_instance = FA()

# Read automaton definition from file
fa_instance.read_from_file("path/to/automaton_definition.txt")

# Check if a sequence is accepted
result = fa_instance.seq_is_accepted("input_sequence")
```

Input file format (EBNF)

```
input_file = states | symbols |  initial_state | final_states | transition_function

letter = "abc..zAB...Z"
digit = "01..9"

states = state | {("," | state)}
state = letter | digit
```

```
symbols = symbol | {("," | symbol)}
symbol = letter | digit

initial_state = state

final_states = state | {("," | state)}

transition_function = transition | {("\n" | transition)}
transition = "(" | state | "," | symbol | ")" | "=" | state
```