https://github.com/alexandra-murariu/FLCD

Documentation

In the scanner implementation, I first read the tokens from tokens.in file and keep them in a list. Then, I scan "words" from the program file (I first split by lines and then by spaces), like this:

```
words = []
with open(self.filename, "r") as f:
    for line in f:
        words.append(line.split())
```

Next, I parse through the list of lists of lines of words, and call the find_tokens function on the current word. This function first further splits the word using this regex:

```
line_data = re.split('("[^ a-zA-Z0-9\"\']")|([^ a-zA-Z0-9\"\'])', string)
```

This regex groups a sequence of letters/digits/quotes/double quotes, i.e in the case of a string represented with double quotes it will preserve its double quotes:

Word "aaa"; -> ['"aaa"', None, ';', '']

Then, we filter the lists to eliminate spaces, empty strings and None values:

```
elements = [el for el in line_data if el is not None and el != '' and el != '
']
```

Then we parse through those elements and check if we have one of the cases "<=", ">=", "!=" etc of composed valid tokens. Basically, at the moment those tokens are being separated and we should unite them in a single value inside the PIF. If the current element and the following element form a valid token, we append the next element to the first one and then delete the second element from the list of tokens of the current "word", i.e:

```
if elements[i] == "=" and elements[i + 1] == "=":
    elements[i] += elements[i + 1]
    del elements[i + 1]
```

After we split the current "word" into elements inside the find_tokens method, the code parses through each of those elements and checks first if they are tokens, then identifiers or constants. If an element is none of those, it means a lexical error is present there so an error is raised with the specific line and element.

- token check:

```
if elem in self.tokens:
```

We just search through the list of tokens and see if the element is inside it.

- identifier check:

```
if elem != "true" and elem != "false" and re.match('^[_a-zA-Z]+[a-zA-Z0-
9_]*$', elem) is not None
```

The above regex means that we look for a sequence of characters that starts with a letter or an underscore (having at least one character) and then a sequence of letters, digits or underscores. The identifier can also be "true" or "false" as we defined the Boolean type.

- constant check:

```
re.match('^\"[a-zA-Z0-9-_ ]+\"$', elem) is not None) or (  # string
re.match('^\'[a-zA-Z0-9\-_ ]\'$', elem) is not None) or (  # char
re.match('^[1-9][0-9]*$|^0$', elem) is not None) or (  # number
re.match('^true|false$', elem) is not None):  # bool
```

A constant can be string, char, number or Boolean.

  - string: the regex signifies a collection of one or more characters (letters, digits, underscore or spaces) within double quotes
  - char: the regex signifies one letter/digit/underscore/space between quotes
  - number: the regex signifies a nonzero digit followed by digits, or just zero
  - bool: the regex signifies accepted values true/false

At the end, the symbol table and PIF are written to file.

Documentation – Symbol Table

The symbol table is implemented with a hash table, using separate chaining.

The HashTable class contains a list of nodes of a given size in the constructor. All nodes are initialized with None. A node is the structure of each value stored inside the HashTable, and has a key, a value (the one that is inserted to the Symbol Table) and a next field, representing the next node inside the linked list at that index inside the HashTable.

The SymbolTable class contains a HashTable of size 23 (it is a common choice for hash tables to have a prime size not near to a power of 2). I also save all values in a list with the only purpose of tabulating the table to output when necessary, not in the hash table format, but in the order in which elements are being added to the table.

I used a hashing algorithm that computes the ascii sum of characters inside the value and returns the sum modulo 23 (size of HashTable, i.e number of containers)

During the add operation, we add the new value to the list if it doesn't already exist. Also, we call the insert operation from the HashTable with the current key (which auto increments at each addition to the HashTable) and the value given. In the insert operation from the HashTable, we compute the hash of the value given (using the hash algotithm mentioned above) and find the index in which we should add the value from the HashTable. We have 3 cases:

1. the value already exists, in which case we return a tuple containing the index inside the HashTable and the key (the location of the value inside the HashTable)
2. the value does not exist, in which we have two subcases:
    a. the node at the index in the HashTable is None (we don't have any values stored at that index), in which case we create a new node with the corresponding key and value, and next=None, and add it to that index
    b. there is at least one node at that index, in which case we go through all existing nodes using the next field of each node, and place the new value in a new node connected to the last inserted one at that index

The insert operation returns the key and index in the hashtable of the added value in each of the cases.

During the find operation, the ST class calls the get operation from the HashTable class. This operation computes the hash of the value to be found and looks inside the container through all the nodes (jumping from node to node with the next field) and searches for a node that has the same value as the one to be found. If this value is found, it returns the node and the index from the HashTable, otherwise it returns None.

The str function returns a string representation of the HashTable.

The st_to_string function returns a table representation of the Symbol Table.