Lab 4: Building LL(1) Parser

This project is a LL(1) parser that takes in grammar rules as input, based on which computes first, follow and parse table. Then it takes as input strings and prints if the strings are valid according to the grammar or not. During the execution, it also checks if the grammar rules provided are valid and based on LL(1) parsing or not.

Description

Here are listed all the methods used for building the project and the assumptions made. Also, sample execution of the program is provided for reference. We first wrote most of the Python and then ported to C++.

Methods

Following is a list of utility methods along with their functionality(' `-> value` ' denotes return type):

- `take_input(grammar, start) -> void`
  - takes in input an empty hashmap `grammar` and a string `start`
  - prompts user to enter grammar rules and accordingly separate non terminals and production rules to store in the hashmap
  - also asks user to mention the `start` symbol
- `eliminate_left_recursion(grammar) -> void`
  - takes in hashmap `grammar` as input and checks for left recursion
  - if left recursion exists, it eliminates it and introduces new non terminals and updates the production rules
- `find_first(non_terminal, grammar, first) -> void`
  - takes in input a `non_terminal` string, hashmap `grammar` and hashmap `first` and computes the first terminals for the `non_terminals`
- `find_follow(non_terminal, grammar, follow, first) -> void`
  - takes in input a `non_terminal` string, hashmap `grammar`, hashmap `follow` and hashmap `first` and computes the follow terminals for the `non_terminals`
- `compute_parse_table(parse_table, grammar, first, follow, non_terminal_to_index, terminal_to_index) -> void`
  - takes in input 2-D matrix `parse_table`, `non_terminal` string, hashmap `grammar`, hashmap `follow` and hashmap `first` and computes the parse table
- `scan(input) -> string`
  - takes in input string which is the input given by user for checking the validity based on grammar rules
  - it converts the numerical part in the input to `n` and alphabetical parts to `i`, denoting `num` and `id` tokens which makes it easier for performing ll(1) parsing on the input
  - it returns the transformed input string

Following is a list of helper functions that proved useful for carrying out above tasks:

- `extract_non_terminal(rule) -> string`
  - takes in input string `rule` and extracts the non terminal that gives the production
  - it returns the found non terminal string
- `extract_production(rule) -> string`
  - takes in input string `rule` and extracts the production part of the rule
  - it returns the production string
- `convert_to_arr(production) -> array`
  - takes in input extracted production string and breaks it into array of individual symbols comprising of terminals and non terminals
  - it returns this array
- `check_left_recursion(non_terminal, productions) -> bool`
  - takes in input string `non_terminal` and 2-D matrix `productions` and returns true if left recursion exists, false otherwise
- `find_non_terminal_to_index_map(grammar, non_terminal_to_index) -> void`
  - takes in input hashmap `grammar` and `non_terminal_to_index` and fills the latter with key-value pairs where key is non terminal and value is an index
  - this hashmap is useful for indexing into the parse table based on non terminal value
- `find_terminal_to_index_map(grammar, terminal_to_index) -> void`
  - takes in input hashmap `grammar` and `terminal_to_index` and fills the latter with key-value pairs where key is terminal and value is an index
  - this hashmap is useful for indexing into the parse table based on terminal value

Assumptions

Following assumptions were made while writing the code, which we tried to eliminate but ran out of time, but are part of future work:

- The grammar rule strings are of the form `non_terminal : production`. Whitespaces between the entities is allowed.
- Non terminals are uppercase and single lettered. Terminals are either lowercase single lettered characters or special characters, except ' `$` '
- `#` represents epsilon

Sample Execution

```
 How many production rules?
9
Enter the rules:
E: E + T
E: E - T
E: T
T: T * F
T: T / F
T: F
F: n
F: i
F: (E)
What is the start symbol
E
-----------Productions-----------
E' => +TE' | -TE' | epsilon
T' => *FT' | /FT' | epsilon
F => n | i | (E)
T => FT'
E => TE'
-------------------------------
-----------Firsts-----------
E' => epsilon, -, +
T' => epsilon, /, *
F => (, i, n
T => n, i, (
E => (, i, n
-------------------------------
-----------Follows-----------
E' => $, )
T' => -, +, $, )
F => *, /, -, $, +, )
T => $, +, -, )
E => $, )
-------------------------------
Enter the string, type 'end' to break
36 * + 49
String accepted
Enter the string, type 'end' to break
36 / -49
String accepted
Enter the string, type 'end' to break
abc + 50
String accepted
Enter the string, type 'end' to break
abc - def
String accepted
Enter the string, type 'end' to break
+ abc + 90
String accepted
Enter the string, type 'end' to break
abc /
Invalid string
Enter the string, type 'end' to break
90 -
Invalid string
Enter the string, type 'end' to break
end
```

Running Locally

Execute the following commands in sequence:

- `chmod a+x cmds.py`

- `./cmds.py build`
- `./build/bin/app`

To clear the build files, execute:

- `./cmds.py clear`

Contributors

- Prateek Jain github@prateek93a
    - Wrote functions for
        - taking grammar input(colab)
        - eliminating left recursion
        - computing follow(colab)
        - computing parse table
        - validating input strings based on the parse table
    - Worked on Readme
- Prem Chandra Singh github@pcsingh
    - Wrote functions for
        - taking grammar input(colab)
        - computing first
        - computing follow(colab)
        - printing first, follow and productions
    - Worked on Readme

(`(colab)` indicates that the worked was done collaboratively on the respective functions)