https://github.com/GamaCatalin/Parser

Gama Marius + Ghetina Vlad

Grammar representation:

line 1: non-terminals separated by space

line 2: terminals separated by space

line 3: start symbol

line 4+: productions on each line following this rule:

non-terminal separated by '->' from symbols

and the symbols separated by " "

ex: S->a b S

CFG check:

1. We take the start symbol and check if it is a non-terminal
    a. If it isn't a non-terminal we throw an error
    b. If it is a non-terminal it continues
2. We take each key from the production dictionary
    a. If any key isn't a non-terminal it throws an error
    b. Then we check every move for each key
3. We take each move from the productions and check it's steps
4. If any step can't be found in non-terminal and terminal symbols it throws an error
5. It returns true

Parser – recursive descendant with father-sibling parsing tree –

https://github.com/GamaCatalin/Parser/blob/main/RDParser.py

The parser has as parameters:

- The current grammar
- The input sequence
- The output file
- The working stack
- The input stack
- Current state:
  - q - normal state
  - b – back state
  - f – final state
  - e – error
- Current index
- The parsing tree
  - Represented as an array of nodes

The current iteration will be written in the output file as such:

{state} {index}
{working stack}
{input stack}

The used methods are:

- Expand
  - The top of the input stack is popped
  - It is added in the working stack as a tuple, having the production index as 0
  - The production for the input is added at the top of the input stack
- Advance
  - The top of the input stack is popped
  - It is added in the working stack
  - The index is incremented
- Momentary-Insuccess
  - Sets the state to 'b'
- Back
  - The top of the working stack is popped

    - It is added in the input stack
    - The index is decremented
- Success
    - Sets the state to ‘f’
- Another-Try
    - The top of the working stack is popped
    - It checks if there are any productions left
        - If there are productions left
            - Sets the state to ‘q’
            - The next production is added in the working stack
            - Changes the production sequence from the input stack
        - If there are no productions left
            - The production sequence is removed
            - The last non-terminal is added to the top of the input stack

The parsing algorithm is implemented as such:

```
While ( state != 'f' and state != 'e') {
    If ( state == 'q' ) {
        If ( input.length() == 0 and index  == sequence.length() ) {
            success()
        }
        elif ( input.length() == 0 ) {
            state = 'e'
            // error
        }
        else {
            if ( input[0] is non-terminal ) {
                expand()
            }
            else {
                if ( index < sequence.length() and input[0] == sequence[index] ){
                    advance()
                }
                else {
                    momentary_insuccess()
                }
            }
        }
    }
    else {
        if ( state == 'b' ) {
            if ( index == 0 and working.length() == 0){
                state = 'e'
                // error
            }
            If ( working[-1] is teminal ) {
                back()
            }
            else {
                another_try()

            }
        }
    }
}
```

Father-sibling parsing tree:

1. We put all the elements in the working stack in the given order
    a. If the element is a tuple (i.e. it is a non-terminal having a given production) we add it and set the production field to the given index
2. We take each element from the tree and we work on it as such:
    a. If it is a terminal it's father is set as the current father if it's father is unasigned
    b. If it is a non-terminal:
        i. We set it's father as the current father
        ii. We set the current father as the index of the element
        iii. We get it's production length
        iv. We get a list of it's {prod_length} following indexes
        v. For each following index we check if it a non terminal, then we get recursively the depth that it goes and push everything down by the computed offset
        vi. We go through the computed indexes and set their father as the current node
        vii. We go through the computed indexes and set their sibling as the next index in the list.

3. We compute the depth of a given element by going through it's production elements and doing the same calculations for each one of it's non-terminal elements.