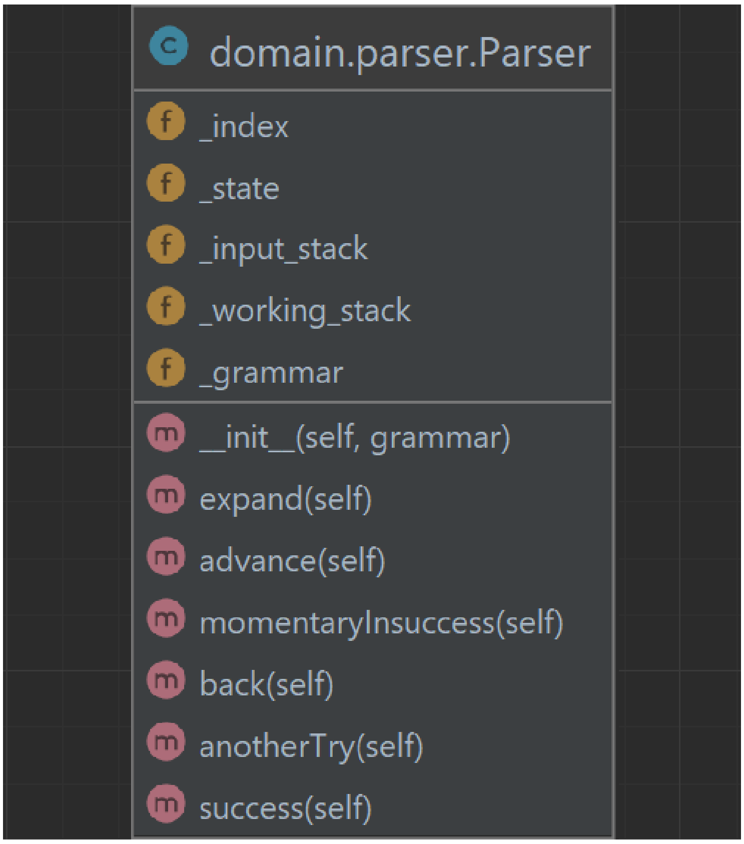
Documentation

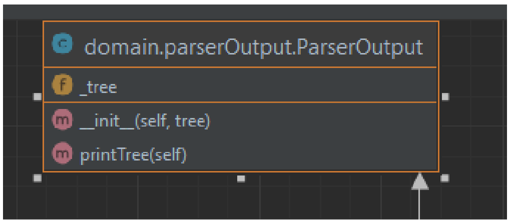
Github links

Diaconu Ana-Maria: https://github.com/DiaconuAna/Formal-Languages-and-Compiler-Design/blob/main/Lab 5 - Parser

Duma Amalia Diana: https://github.com/AmaliaDuma/Formal-Languages-and-Compiler-Design/tree/main/Labs/Lab_4 - Parser

Class diagram





Parser: class attributes

- `working stack`: working stack alpha which stores the way the parse is built
- `input_stack`: input stack beta which is a part of the tree to be built
- `state`: state of the parsing which can take one of the following values:
  - q = normal state

    - b = back state
    - f = final state - corresponding to success: w ε L(G)
    - e = error state – corresponding to insuccess: w ∉ L(G)

- `i`: position of current symbol in input sequence
- `grammar`: grammar of the language for which we will perform the sequence check
- `sequence`: sequence read from input file (seq.txt for g1 and PIF.out for g2)
- `out_file`: name of the output file

ParserOutput: class attributes

- `tree`: a list that contains `Item` entries

Item: class attributes

- `father`: the index of the father of crt element
- `sibling`: index of whose left sibling crt element is
- `value`: symbol of crt element
- `production`: number of production

Methods

▼ `expand()`

```
Occurs when the head of the stack is a non-terminal
(q,i,α, Aβ) ⊢ (q,i,αA1,γ1β)
where:
A →γ1 |γ2 | … represents the productions corresponding to A
1 = first prod of A

Steps:

1. pop A from the input stack beta
2. add A1 to the working stack alpha
3. Get the first production of A
4. Add the corresponding production to the input stack beta
```

▼ `advance()`

```
WHEN: head of input stack is a terminal = current symbol from input
(q,i,α, aiβ) ⊢ (q,i+1,αai,β)

Steps:

1. get the top of the input stack
2. add it to the working stack
3. increase index i
```

▼ `momentaryInsuccess()`

```
WHEN: head of input stack is a terminal ≠ current symbol from input
(q,i,α, aiβ) ⊢ (b,i,α, aiβ)

Steps:

1.State becomes back.
```

▼ `back()`

```
WHEN: head of working stack is a terminal
(b,i,αa,β) ⊢ (b,i-1,α, aβ)

Steps:

1. get the last element from the working stack
2. add it back to the input stack
3. decrease index
```

▼ `anotherTry()`

```
WHEN: head of working stack is a nonterminal
        (b,i, α Aj, γj β) ⊢ (q,i, αAj+1, γj+1β) , if ∃ A → γj+1
                            (b,i, α, Aβ), otherwise with the exception
                            (e,i, α,β), if i=1, A =S, ERROR

        Steps:

        1. get the top of the working stack: tuple of form (non_terminal, production_nr)
        2. check if we have more productions for that non-terminal
            2.1. update the state as 'q': normal state
            2.2. create a new tuple consisting of (non_terminal, production_nr+1) and add it to the working stack
                 (moving on to the next production)
            2.3. Update the top of input stack with the new production: delete old one and replace it
            2.4. Slice the list to delete last production
            2.5. Insert the new one on top
        3. if there are no more productions for the current terminal we check the following condition:
           (e,i, α,β), if i=1, A =S, ERROR
        4. otherwise, delete the last production from the working stack and put the corresponding non-terminal in the
           input stack
```

▼ `success()`

```
(q,n+1,α,ε) ⊢ (f,n+1,α,ε)

Steps:

1. Mark the state as final
```

▼ `createParsingTree()`

```
  Creates the parsing tree by iterating over the working stack and for every
element present it adds a new entry, then it updates the father-sibling
relationships.
```

▼ `get_length_depth()`

```
Input parameter:
   index -> index of the current production in the working stack

Gets the length of the used production by going in depth
```

▼ `parsingStrategy(w)`

```
Input parameter:
    w -> sequence to be parsed

Parse a sequence using the descendent recursive parsing from the lecture
```

▼ `read_sequence(sequence_file)`

Reads the sequence from an input file:

- seq.txt for g1 - sequence elements are placed on different lines
- PIF.out for g2 - sequence elements extracted from PIF

▼ `printCurrentConfigurationToFile()`

Writes the data related to the current configuration (state, index, working and input stacks) to the output file

▼ `write_in_output_file(message, final)`

Writes a given message to the output file. If final is marked as `True`, a message displaying `Sequence accepted` is written to the file.

See `out1.txt`, `out2.txt`, `g1.txt`, `g2.txt`, `seq.txt` and `PIF.out` for more input and output details.