Documentation Parser

Link for our project:

https://github.com/AnastasiaSusciuc/UBB/tree/main/Anul 3/Sem5/FLCD/Labs/Lab6p

Authors:

Susciuc Anastasia & Tise Tudor, group 937/1

Requirements:

Implement a parser algorithm based on LL(1) method.

The representation of the parsing tree should be a table using father-sibling implementation.

Example

Input:

grammar

```
S A B C D
a + * ( )
S
S -> B A
A -> + B A
A -> E
B -> D C
C -> * D C
C -> E
D -> ( S )
D -> a
```

seq

`a + ( a * a )`

Output:

First Set

```
S: ( a
A: + E
B: ( a
C: * E
D: ( a
a: a
+: +
*: *
(: (
): )
```

FOLLOW SET

```
S: ) $
A: ) $
B: ) + $
C: ) + $
D: ) + * $
```

TABLE

```
('S', 'a') -> (['B A'], 1)
('S', '(') -> (['B A'], 1)
('A', '+') -> (['+ B A'], 2)
('A', ')') -> (['E'], 3)
('A', '$') -> (['E'], 3)
('B', 'a') -> (['D C'], 4)
('B', '(') -> (['D C'], 4)
('C', ' ') -> ([' D C'], 5)
('C', ')') -> (['E'], 6)
('C', '+') -> (['E'], 6)
('C', '$') -> (['E'], 6)
('D', '(') -> (['( S )'], 7)
('D', 'a') -> (['a'], 8)
('a', 'a') -> ('pop', -1)
('+', '+') -> ('pop', -1)
(' ', ' ') -> ('pop', -1)
('(', '(') -> ('pop', -1)
(')', ')') -> ('pop', -1)
('$', '$') -> ('acc', -1)
```

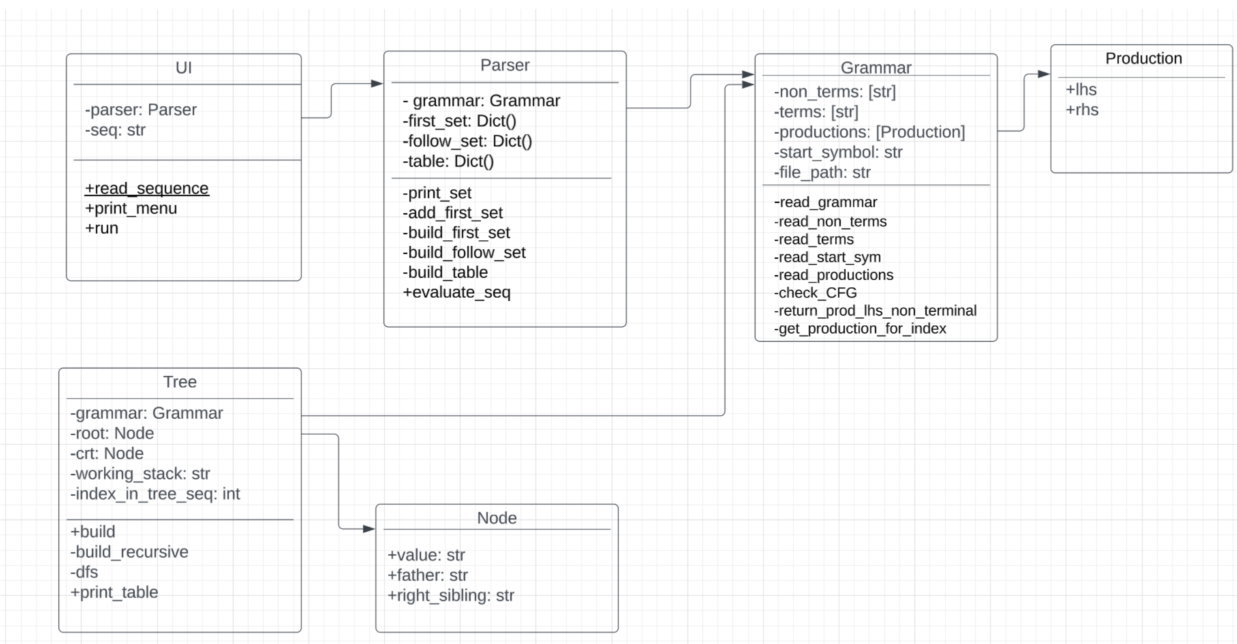
Tree

```
Index: 1 | Value: S | Father: None | Left_Sibling: None
Index: 2 | Value: B | Father: 1 | Left_Sibling: None
Index: 3 | Value: A | Father: 1 | Left_Sibling: 2
Index: 4 | Value: D | Father: 2 | Left_Sibling: None
Index: 5 | Value: C | Father: 2 | Left_Sibling: 4
Index: 6 | Value: a | Father: 4 | Left_Sibling: None
Index: 7 | Value: E | Father: 5 | Left_Sibling: None
Index: 8 | Value: + | Father: 3 | Left_Sibling: None
Index: 9 | Value: B | Father: 3 | Left_Sibling: 8
Index: 10 | Value: A | Father: 3 | Left_Sibling: 9
Index: 11 | Value: D | Father: 9 | Left_Sibling: None
```

```
Index: 12 | Value: C | Father: 9 | Left_Sibling: 11
Index: 13 | Value: ( | Father: 11 | Left_Sibling: None
Index: 14 | Value: S | Father: 11 | Left_Sibling: 13
Index: 15 | Value: ) | Father: 11 | Left_Sibling: 14
Index: 16 | Value: B | Father: 14 | Left_Sibling: None
Index: 17 | Value: A | Father: 14 | Left_Sibling: 16
Index: 18 | Value: D | Father: 16 | Left_Sibling: None
Index: 19 | Value: C | Father: 16 | Left_Sibling: 18
Index: 20 | Value: a | Father: 18 | Left_Sibling: None
Index: 21 | Value: * | Father: 19 | Left_Sibling: None
Index: 22 | Value: D | Father: 19 | Left_Sibling: 21
Index: 23 | Value: C | Father: 19 | Left_Sibling: 22
Index: 24 | Value: a | Father: 22 | Left_Sibling: None
Index: 25 | Value: E | Father: 23 | Left_Sibling: None
Index: 26 | Value: E | Father: 17 | Left_Sibling: None
Index: 27 | Value: E | Father: 12 | Left_Sibling: None
Index: 28 | Value: E | Father: 10 | Left_Sibling: None
```

Implementation:

UML Diagram:



The **Grammar** class has a field for each (non_terms, terms, productions, start_symbol) set of the grammar, namely *terminals*, *non terminals*, *productions* and *a starting symbol*.

The set of productions P is kept as a list of **Production**, which has two attributes, namely the left-handside of the production and the right-handside, both being list of string (a string might represent a terminal or a non-terminal).

In the **Parser** class, most of the methods are for file parsing. Since we are implementing the LL(1) algorithm, we also implemented the first and follow algorithms.

The **first** algorithm builds a set for each non-terminal that contains all terminals from which we can start a sequence, starting from that given non-terminal.
The **follow** builds a set for each non-terminal basically returns the “first of what’s after”, namely all the non-terminals into which we can proceed from the given non-terminal.

Having these 2 sets built for each non-terminal (and for terminals also, but those are trivial), we proceed to build the **LL(1) parse table**. We follow the rules given in the lecture: we build a table that has as rows all non-terminals + terminals, and as rows, all terminals, plus the “$” sign in both rows and columns. We then follow the rules given in the seventh lecture, slide 9.

Having the parse table built, the next step is **parsing** a given input sequence with the LL(1) parsing algorithm, following the push/pop rules from slide 13 from the same lecture 7. This will build an output which we **recursively build a tree** – we start from the root, and then we take care of its first child and then its right sibling.
The last step is iterating through the tree and **printing** the obtained data.

Seminar example that helped us:

$\boxed{LL1}$

$G = (\{S, A, B, C, D\}, \{a, +, *, (,)\}, P, S)$

$P:$
(1) $S \to BA$
(2) $A \to +BA$
(3) $A \to \varepsilon$
(4) $B \to DC$
(5) $C \to *DC$
(6) $C \to \varepsilon$
(7) $D \to (S)$
(8) $D \to a$

$w = a * (a + a)$

I First and Follow

$First(\alpha) = First(x_1 x_2 \dots x_n) = First(x_1) \oplus First(x_2) \oplus \dots First(x_n)$

$\alpha = x_1 x_2 \dots x_n, \quad x_i \in V, \quad i = \overline{1..n}$

First:

	F_0	F_1	F_2	F_3 = F_2 = First
S	∅	∅	(, a	(, a
A	+, ε	+, ε	+, ε	+, ε
B	∅	(, a	(, a	(, a
C	*, ε	*, ε	*, ε	*, ε
D	(, a	(, a	(, a	(, a

DC $\{a, (\}$ $\{*, \varepsilon\}$

a*
a
(*
(

FOLLOW:

	L_0	L_1	L_2	L_3	=	L_4 = follow
S	ε	ε, (	ε, (	ε, (		ε, (
A	∅	ε	ε,)	ε,)		ε,)
B	∅	+, ε	+, ε,)	+, ε,)		+, ε,)
C	∅	∅	+, ε	+, ε,)		+, ε,)
D	∅	*	*,)	*, +, ε,)		*, +, ε,)

first(S) = {(, a}
first(A) = {ε, +}
first(B) = {(, a}
first(C) = {ε, *}
first(D) = {(, a}

follow(S) = {ε,)}
follow(A) = {ε,)}
follow(B) = {+, ε,)}
follow(C) = {+, ε,)}
follow(D) = {*, +, ε,)}

II LL(1) Parsing table

	a	+	*	(	)	$
S	BA, 1			BA, 1		
A		+BA, 2			ε, 3	ε, 3
B	DC, 4			DC, 4		
C		ε, 6	*DC, 5		ε, 6	ε, 6
D	a, 8			(S), 7		
a	pop					
+		pop				
*			pop			
(				pop		
)					pop	
$						

III Parsing of w

i.e. $(w\$, S\$, \varepsilon)$

f.c $(\$, \$, \alpha)$