LL(1) Parser

Student: Alexandrescu Andrei-Robert, 931/1

Current lab: [Repository Link]

1. Parser

Class **Parser** is responsible for implementing the LL(1) parser. It contains the first and follow methods that are used to obtain the *first* and *follow* tables. These two methods are implemented in the same way they were specified in the lectures.

The class Parser keeps only the final version of the *first* and *follow* tables. That means, the steps that lead to the final version of *first* and *follow* are not stored because they are not relevant for the problem.

In order to write a cleaner implementation for the *first* and *follow* methods, some methods were added to the Grammar class. These methods are:

- *getIsTerminal*, *getIsNonTerminal*, *getIsEpsilon*, *getIsStartingSymbol*: these are used for checking whether a given symbol belongs to one of the above-mentioned classes
- *getProductionsWithNonTerminalOnRHS*: this method retrieves a vector of productions containing a given symbol in the right-hand side. This method is used in the *follow* algorithm.
- *splitRHSOnNonTerminal*: this method takes a production and a non-terminal (which must belong to the right-handside of the production). It splits the production into two parts, denoted by alpha and gamma. Alpha is the sequence of symbols before the non-terminal and gamma is the sequence after.

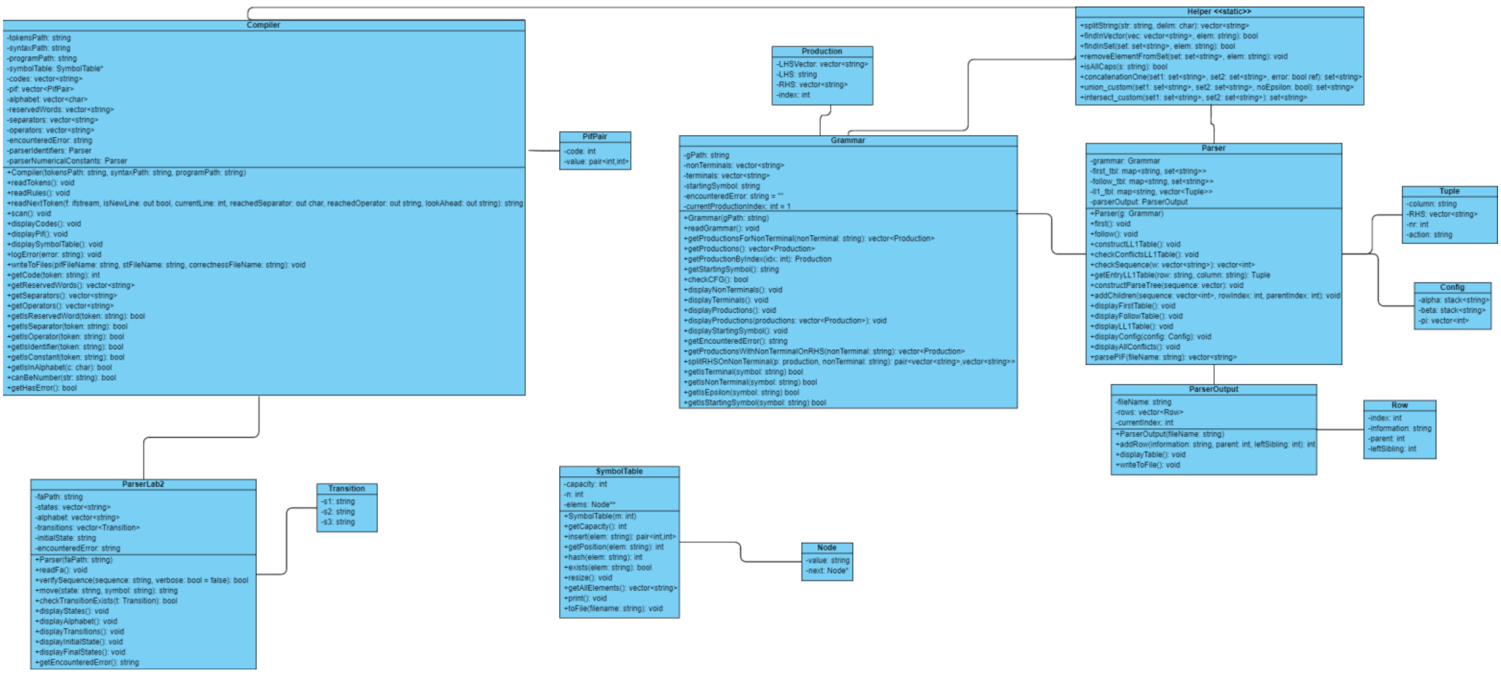
The set operations (union, concatenation of length one) were implented in the **Helper** class.

The method for constructing the LL(1) parse table is *constructLL1Table*. Method *checkSequence* receives a sequence of terminal symbols and checks whether the sequence belongs to the language generated by grammar G or not. It returns a vector of integers representing the indexes of the productions that must be performed to solve the sequence.

2. ParserOutput

Class **ParserOutput** is responsible for storing the parse tree (or syntax tree) generated by method *constructParseTree*. This method requires the sequence of indexes from *checkSequence*.

The class diagram (with the integration with the analyser) becomes:



3. Experiments:

The experiments section is divided into three parts:

a) FIRST and FOLLOW
b) LL(1) Table, Sequence parsing, Syntax Tree
c) Syntax Tree

a)

The First And Follow tables were implemented using the algorithm from the lecture, with a small exception: the Follow algorithm was taken from the manual because it was mor explicit regarding a condition with Epsilon.

The results of the First and Follow algorithms can be visualised below in two examples, on grammar g8 and g22

Example 1 *(g8.in)*

S A B C D
a b c
5
S -> a A b | B A
A -> a A | c A | c
B -> D C
D -> Epsilon | b
C -> c
S

```
[First table ...]
A: a c
B: b c
C: c
D: Epsilon b
S: a b c
a: a
b: b
c: c
```

```
[Follow table ...]
A: b Epsilon
B: a c
C: a c
D: c
S: Epsilon
```

First table:

	F_0	F_1	F_2	F_3=F_2
S	a	a	a, b, c	
A	a, c	a, c	a, c	
B	∅	b, c	b, c	
C	c	c	c	
D	ε, b	ε, b	ε, b	

Follow table:

	L_0	L_1	L_2	L_3=L_2
S	ε	ε	ε	
A	∅	b, ε	b, ε	
B	∅	a, c	a, c	
C	∅	∅	a, c	
D	∅	c	c	

Example 2 *(g22.in)*

S A
a b c
2
S -> A a | a
A -> b A | c
S

```
[First table ...]
A: b c
S: a b c
a: a
b: b
c: c
```

```
[Follow table ...]
A: a
S: Epsilon
```

First table:

	F_0	F_1	F_2=F_1
S	a	a, b, c	
A	b, c	b, c	

Follow table:

	L_0	L_1	L_2=L_1
S	ε	ε	
A	∅	a	

b) In the following examples we compute both the First and Follow tables, together with the LL(1) table. Then we parse a sequence and write the parse/syntax tree.

Example 3 *(g3.in)*

S A B C
a b c d
4
S -> A a | B B
A -> b A | c
B -> C a
C -> d
S

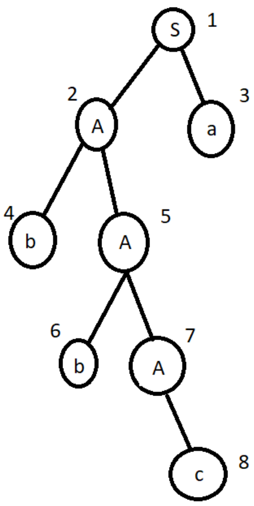
```
[First table ...]
A: b c
B: d
C: d
S: b c d
a: a
b: b
c: c
d: d
```

```
[Follow table ...]
A: a
B: d
C: a
S: Epsilon
```

```
[LL1 table ...]
$: $: acc
A: b: (bA,3) c: (c,4)
B: d: (Ca,5)
C: d: (d,6)
S: b: (Aa,1) c: (Aa,1) d: (BB,2)
a: a: pop
b: b: pop
c: c: pop
d: d: pop
```

Sequence: **bbca**

```
[Checking sequence ...]
(bbca$ , S$ , )
[PUSH]
(bbca$ , Aa$ , 1)
[PUSH]
(bbca$ , bAa$ , 13)
[POP]
(bca$ , Aa$ , 13)
[PUSH]
(bca$ , bAa$ , 133)
[POP]
(ca$ , Aa$ , 133)
[PUSH]
(ca$ , ca$ , 1334)
[POP]
(a$ , a$ , 1334)
[POP]
($ , $ , 1334)
[ACCEPT]
Sequence: 1 3 3 4
```




```
[Table ...]
Id  Info  Parent  Left-Sibling
1   S   0   0
2   A   1   0
3   a   1   1
4   b   2   0
5   A   2   1
6   b   5   0
7   A   5   1
8   c   7   0
[... done]
```

Example 4 *(g4.in)*

S B C
a b c d
3
S -> B b | C d
B -> a B | Epsilon
C -> c C | Epsilon
S

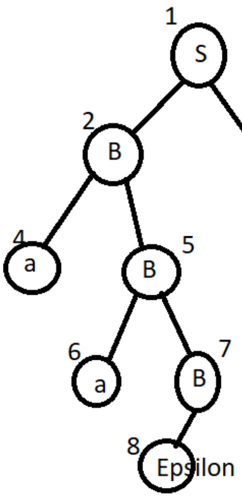
```
[First table ...]
B: a Epsilon
C: c Epsilon
S: a b c d
a: a
b: b
c: c
d: d
```

```
[Follow table ...]
B: b
C: d
S: Epsilon
```

```
[LL1 table ...]
$: $: acc
B: a: (aB,3) b: (Epsilon,4)
C: c: (cC,5) d: (Epsilon,6)
S: a: (Bb,1) b: (Bb,1) c: (Cd,2) d: (Cd,2)
a: a: pop
b: b: pop
c: c: pop
d: d: pop
```

Sequence: **aab**

```
[Checking sequence ...]
(aab$ , S$ , )
[PUSH]
(aab$ , Bb$ , 1)
[PUSH]
(aab$ , aBb$ , 13)
[POP]
(ab$ , Bb$ , 13)
[PUSH]
(ab$ , aBb$ , 133)
[POP]
(b$ , Bb$ , 133)
[PUSH]
(b$ , b$ , 1334)
[POP]
($ , $ , 1334)
[ACCEPT]
Sequence: 1 3 3 4
```




```
[Table ...]
Id   Info   Parent   Left-Sibling
1    S   0   0
2    B   1   0
3    b   1   2
4    a   2   0
5    B   2   4
6    a   5   0
7    B   5   6
8    Epsilon   7   0
[... done]
```

Example 5 *(g1.in)*

S
a b c
1
S -> a S b S | c
S

```
[First table ...]
S: a c
a: a
b: b
c: c
[... done]
```

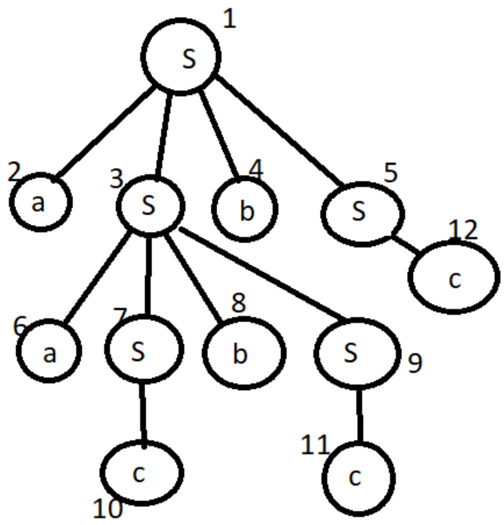
```
[Follow table ...]
S: Epsilon b
[... done]
```

```
[LL1 table ...]
$: $: acc
S: a: (aSbS,1) c: (c,2)
a: a: pop
b: b: pop
c: c: pop
[... done]
```

Sequence: **aacbcbc**

```
[Checking sequence ...]
(aacbcbc$ , S$ , )
[PUSH]
(aacbcbc$ , aSbS$ , 1)
[POP]
(acbcbc$ , SbS$ , 1)
[PUSH]
(acbcbc$ , aSbSbS$ , 11)
[POP]
(cbcbc$ , SbSbS$ , 11)
[PUSH]
(cbcbc$ , cbSbS$ , 112)
[POP]
(bcbc$ , bSbS$ , 112)
[POP]
(cbc$ , SbS$ , 112)
[PUSH]
(cbc$ , cbS$ , 1122)
[POP]
(bc$ , bS$ , 1122)
[POP]
(c$ , S$ , 1122)
[PUSH]
(c$ , c$ , 11222)
[POP]
($ , $ , 11222)
[ACCEPT]
Sequence: 1 1 2 2 2
```

```
[Table ...]
Id  Info  Parent  Left-Sibling
1   S   0   0
2   a   1   0
3   S   1   2
4   b   1   3
5   S   1   4
6   a   3   0
7   S   3   6
8   b   3   7
9   S   3   8
10  c   7   0
11  c   9   0
12  c   5   0
[... done]
```



c) This last section of experiments presents the parsing of a program using the cusom grammar. We considered the simplest program in order to obtain results that are trivial to parse.

p4.in

```
START
PRINT("Hello world")
FINISH
```

```
Id  Info  Parent  Left-Sibling
1   program  0  0
2   START  1  0
3   cmds  1  2
4   FINISH  1  3
5   cmd  3  0
6   cmdsconf  3  5
7   simplecmd  5  0
8   printcmd  7  0
9   PRINT  8  0
10  (  8  9
11  expressionprint  8  10
12  )  8  11
13  factorprint  11  0
14  expressionprintconf  11  13
15  constant  13  0
16  Epsilon  14  0
17  Epsilon  6  0
```