LL(1) Grammars & LL(1) Parsing

Exercise 1

Consider the following grammar for balanced parentheses:

```
S ::= B EOF
B ::= ε | B ( B )
```

Question 1.1

Compute NULLABLE, FIRST and FOLLOW for each non-terminal.

Only `B` is `NULLABLE`.

```
FIRST(S) = { ( , EOF }
FIRST(B) = { ( }

FOLLOW(S) = { }
FOLLOW(B) = { EOF , ( , ) }
```

Question 1.2

Build the LL(1) parsing table for the grammar. Is the grammar LL(1)?

	(	)	EOF
S	1		1
B	**1, 2**	1	1

No, the grammar is not LL(1).

Question 1.3

Build the LL(1) parsing table for the following grammar. Is the grammar LL(1)?

```
S ::= B EOF
B ::= ε | ( B ) B
```

Only `B` is `NULLABLE`.

```
FIRST(S) = { ( , EOF }
FIRST(B) = { ( }
```

```
FOLLOW(S) = { }
FOLLOW(B) = { EOF , ) }
```

	(	)	EOF
S	1		1
B	2	1	1

Yes, the grammar is LL(1).

Question 1.4

Run the LL(1) parsing algorithm on the following input strings. Show the derivation you obtain, or the derivation up until the error.

```
( ( ) ( ) ) EOF

S                          ==>
B EOF                      ==>
( B ) B EOF                ==>
( ( B ) B ) B EOF          ==>
( ( ) B ) B EOF            ==>
( ( ) ( B ) B ) B EOF      ==>
( ( ) ( ) B ) B EOF        ==>
( ( ) ( ) ) B EOF ==>
( ( ) ( ) ) EOF
```

```
( ) ) ( EOF

S                  ==>
B EOF              ==>
( B ) B EOF        ==>
( ) B EOF
```

Question 1.5

Show that the second grammar describes the language of balanced parenthesis.
To do so, show that:

1) Every parse tree of the grammar yields a balanced parenthesis string.
2) For every balanced parenthesis string, there exists a parse tree.

Point 1) is trivially shown by induction on the parse tree. Point 2) is trickier. The crucial observation is that every string of balanced parentheses has a prefix (possibly spanning the entire string) that starts with a "(" and ends with a ")" and is itself a balanced string of parenthesis. The rest of the string – after the prefix – is also a balanced parentheses string. We can therefore prove this property by induction on the size of the balanced string.

Exercise 2

Question 2.1

Build a LL(1) grammar for expressions consisting of variables, infix binary addition and multiplication (with usual priorities), and parentheses. Note that we do not care about associativity of binary operations at the level of the parse tree.

Build the LL(1) parsing table.

```
S  ::= T EOF
T  ::= F RT
RT ::= + T | ε
F  ::= B RF
RF ::= * F | ε
B  ::= id | ( T )
```

Or, also valid:

```
S  ::= T EOF
T  ::= F RT
RT ::= + F RT | ε
F  ::= B RF
RF ::= * B RF | ε
B  ::= id | ( T )
```

Computing `NULLABLE`, `FIRST`, `FOLLOW`, (works for both grammars).

Only `RT` and `RF` are `NULLABLE`.

```
FIRST(B) = { id , ( }
FIRST(RF) = { * }
FIRST(F) = { id , ( }
FIRST(RT) = { + }
FIRST(T) = { id , ( }
FIRST(S) = { id , ( }

FOLLOW(S) = { }
```

```
FOLLOW(T) = { EOF , ) }
FOLLOW(RT) = { EOF , ) }
FOLLOW(F) = { + , EOF , ) }
FOLLOW(RF) = { + , EOF , ) }
FOLLOW(B) = { * , + , EOF , ) }
```

Parsing table (works for either of the two grammars).

	id	+	*	(	)	EOF
S	1			1		
T	1			1		
RT		1			2	2
F	1			1		
RF		2	1		2	2
B	1			2		

Question 2.2

Parse the following input strings using your parsing table. Show the derivations and parse trees.

Derivations shown for the second grammar.

id + id * (id + id) EOF

```
S                                     ==>
T EOF                                 ==>
F RT EOF                              ==>
B RF RT EOF                           ==>
id RF RT EOF                          ==>
id RT EOF                             ==>
id + F RT EOF                         ==>
id + B RF RT EOF                      ==>
id + id RF RT EOF                     ==>
id + id * B RF RT EOF                 ==>
id + id * ( T ) RF RT EOF             ==>
id + id * ( F RT ) RF RT EOF          ==>
id + id * ( B RF RT ) RF RT EOF       ==>
```

```
id + id * ( id RF RT ) RF RT EOF       ==>
id + id * ( id RT ) RF RT EOF          ==>
id + id * ( id + F RT ) RF RT EOF      ==>
id + id * ( id + B RF RT ) RF RT EOF   ==>
id + id * ( id + id RF RT ) RF RT EOF  ==>
id + id * ( id + id RT ) RF RT EOF     ==>
id + id * ( id + id ) RF RT EOF        ==>
id + id * ( id + id ) RT EOF           ==>
id + id * ( id + id ) EOF
```

id * id * id EOF

```
S                          ==>
T EOF                      ==>
F RT EOF                   ==>
B RF RT EOF                ==>
id RF RT EOF               ==>
id * B RF RT EOF           ==>
id * id RF RT EOF          ==>
id * id * B RF RT EOF      ==>
id * id * id RF RT EOF     ==>
id * id * id RT EOF        ==>
id * id * id EOF
```

((id)) EOF

```
S                                       ==>
T EOF                                   ==>
F RT EOF                                ==>
B RF RT EOF                             ==>
( T ) RF RT EOF                         ==>
( F RT ) RF RT EOF                      ==>
( B RF RT ) RF RT EOF                   ==>
( ( T ) RF RT ) RF RT EOF               ==>
( ( F RT ) RF RT ) RF RT EOF            ==>
( ( B RF RT ) RF RT ) RF RT EOF         ==>
( ( id RF RT ) RF RT ) RF RT EOF        ==>
( ( id RT ) RF RT ) RF RT EOF           ==>
( ( id ) RF RT ) RF RT EOF              ==>
( ( id ) RT ) RF RT EOF                 ==>
```

```
( ( id ) ) RF RT EOF            ==>
( ( id ) ) RT EOF               ==>
( ( id ) ) EOF
```

Exercise 3

Consider the following grammar for roman numerals 0 to 9:

$$
\begin{aligned}
S &::= N\ \mathbf{EOF} \\
N &::= \mathbf{i}\ A \mid I_3 \mid \mathbf{v}\ I_3 \mid \varepsilon \\
A &::= \mathbf{v} \mid \mathbf{x} \\
I_3 &::= I_2\ \mathbf{i} \mid \varepsilon \\
I_2 &::= I_1\ \mathbf{i} \mid \varepsilon \\
I_1 &::= \mathbf{i} \mid \varepsilon
\end{aligned}
$$

Question 3.1

Is the grammar ambiguous? In other words, do there exist multiple parse trees for the same words ?

No, the grammar is not ambiguous. For every non-terminal, the language of each alternatives is disjoint from the others.

Question 3.2

Is the grammar LL(1)? To motivate your answer, build an LL(1) parsing table. If there are conflicts, discuss why each conflict appears.

The grammar is not LL(1). For instance, in the non-terminal `N`*, the entry for terminal* **`i`** *will contain both 1 and 2, since both alternatives can start with* **`i`**.

Question 3.3

Assuming the above is not LL(1), build a LL(1) grammar for the same language. Show it is LL(1) by building a parsing table.

$$
\begin{aligned}
S &::= N\ \mathbf{EOF} \\
N &::= \mathbf{i}\ A \mid \mathbf{v}\ I_3 \mid \varepsilon \\
A &::= I_2 \mid \mathbf{v} \mid \mathbf{x} \\
I_3 &::= \mathbf{i}\ I_2 \mid \varepsilon \\
I_2 &::= \mathbf{i}\ I_1 \mid \varepsilon \\
I_1 &::= \mathbf{i} \mid \varepsilon
\end{aligned}
$$

N, A, I_3, I_2 and I_1 are NULLABLE.

$\text{FIRST}(I_1) = \text{FIRST}(I_2) = \text{FIRST}(I_3) = \{\ \mathbf{i}\ \}$

FIRST(A) = { **i** , **v** , **x** }
FIRST(N) = { **i** , **v** }
FIRST(S) = { **i** , **v** , **EOF** }

FOLLOW(S) = { }
FOLLOW(N) = FOLLOW(A) = FOLLOW(I_1) = FOLLOW(I_2) = FOLLOW(I_3) = { **EOF** }

	i	v	x	EOF
S	1	1		1
N	1	2		3
A	1	2	3	2
I_3	1			2
I_2	1			2
I_1	1			2