Computer Language Processing

Exercise Sheet 03 - Solutions

October 25, 2022

Exercises **1** and **2** are taken from Basics of Compiler Design, and you can find the solutions here.

Exercise 3

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

   Only B is NULLABLE.

   FIRST(S) = { (, **EOF** }

   FIRST(B) = { (}

   FOLLOW(S) = { }

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

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)

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 }

   FIST(B) = { (}

   FOLLOW(S) = { }

   FOLLOW(B) = { EOF ,)}

	(	)	EOF
S	1		1
B	2	1	1

Yes the grammar is LL(1)

4. Run the LL(+) 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**

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

   (a) Every parse tree of the grammar yields a balanced parenthesis string.
   This point is trivially shown by induction on the parse tree.

   (b) For every balanced parenthesis string, there exists a parse tree.
   This point 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 4

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		

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 9 rf rt **EOF** ==>

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

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

((**id**)) RT **EOF** ==>

((id)) EOF ==>