EX1 A possible regular definition is the following one:

letter → a | b | ⋯ | z | A | B | ⋯ | Z

digit → 0 | 1 | ⋯ | 9

digitnoz → 1 | 2 | ⋯ | 9

name → letter · (letter | digit)*

paramnum → digitnoz · (digit)*

parameter → { paramnum }

commands → (\ | !) · name · (ε | parameter)

EX2 1) The language generated by the grammar is the following one:

$$L(S) = \{ a^{2n+1} x b^n d \mid n \geq 0,\ x \in \{c, d\} \}$$

$$\cup$$

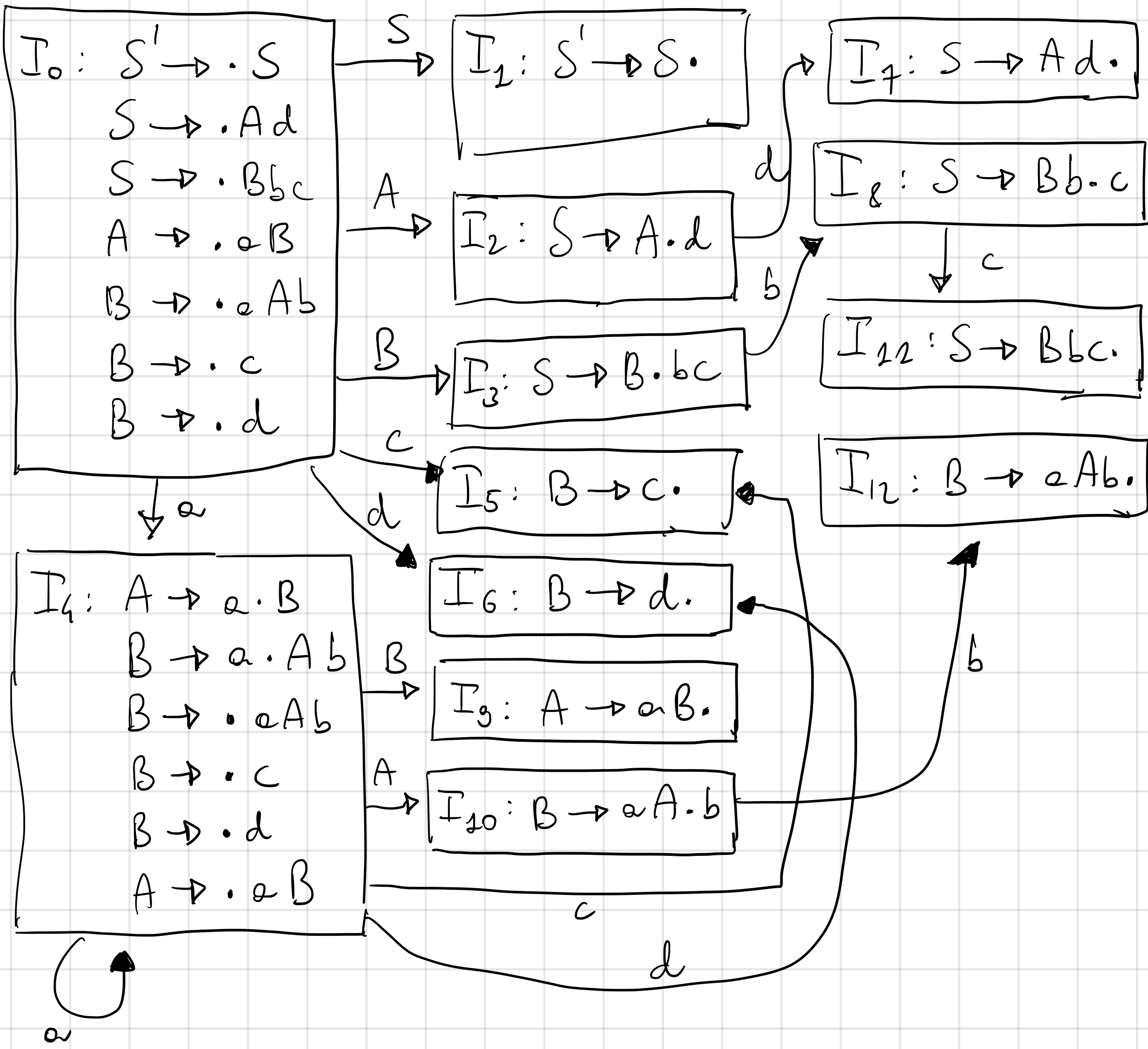
$$\{ a^{2n} x b^n bc \mid n \geq 0,\ x \in \{c, d\} \}$$

2) To determine if the grammar is LR(1) let us try first to construct an SLR table.

Let us construct the collection of items LR(0) (in the following page)

As usual the production S' → S is added.

The numbering of the productions starts with 0 on S' → S



To construct the SLR table we need to calculate the FOLLOWS and to check if there are conflicts

FOLLOWS(S') = { $ }

FOLLOWS(S) = { $ }

FOLLOWS(A) = { d, b }

FOLLOWS(B) = { b, d }

The SLR table is in the following page

STATE	a	b	c	d	$	S	A	B
0	s4		s5	s6		1	2	3
1					acc			
2				s7				
3		s8						
4	s4		s5	s6			10	9
5		r5		r5				
6		r6		r6				
7					r1			
8			s11					
9		r3		r3				
10		s12						
11					r2			
12		r4		r4				

The table is not multiply-defined, thus the grammar is SLR. This means that the grammar is also LR(1).

3) To determine all the valid items for the viable prefix aaaaa it is sufficient to apply the theoretical result that says that all the valid items are those in the state of the collection of LR(0) items that is reached from I_0 (the initial state) considering the collection as an automaton. In our case:

$$I_0 \xrightarrow{a} I_4 \xrightarrow{a} I_4 \xrightarrow{a} I_4 \xrightarrow{a} I_4 \xrightarrow{a} I_4$$

Thus, all the valid items for aaaaa are

$$\begin{cases} A \to a \cdot B \\ B \to a \cdot Ab \\ B \to \cdot aAb \\ B \to \cdot c \\ B \to \cdot d \\ A \to \cdot aB \end{cases}$$

EX 4 Let us first define a suitable grammar for the language. The associated SDD must be suitable for bottom-up parsing, so the grammar must be concieved to have all synthesized attributes.

To implement the associativity and precedence rules in the expressions we can follow the standard approach. To define the list of expressions it is better to use a right recursion.

A suitable grammar is the following:

$S \rightarrow E ; S \mid E$ → (right-recursive list)

$E \rightarrow T \otimes E \mid T$ → (E is right-recursive, so the operator $\otimes$ will be right-associative. Moreover, since T will generate the $\oplus$ operator, then $\oplus$ will have precedence on $\otimes$)

$T \rightarrow F \oplus T \mid F$ → idem for T, $\oplus$ w.r.t. F.

$F \rightarrow \underline{id} \mid \underline{num} \mid (E)$

Let us suppose that the tokens <u>id</u> and <u>num</u> have an attribute "size" that is already calculated by the lexical analyzer.

Let us define an attribute "size", synthesized, for the non-terminal symbols F, T, E. They hold the size of the corresponding sub-trees. In case of non-terminal S,

we define two synthesized attributes:

- minsize, integer, which is the size of the shortest expression in S; it is meaningful only if the other attribute is true
- increasing, boolean; it is the requested attribute. It is true if and only if the sub-list has expressions of strictly increasing size and the subexpression has size strictly shorter than the minsize of the sub-list.

The SDD is the following one:

$S \to E ; S_1$	S.increasing := (S_1.increasing AND E.size < S_1.minsize) S.minsize := E.size
$S \to E$	S.increasing := true S.minsize := E.size
$E \to T \otimes E_1$	E.size := T.size + E_1.size + 1
$E \to T$	E.size := T.size
$T \to F \oplus T_1$	T.size := F.size + T_1.size + 1
$T \to F$	T.size := F.size
$F \to$ id	F.size := id.size
$F \to$ num	F.size := num.size
$F \to (E)$	F.size := E.size + 2