Syntax Analysis - LR(k)

Prof. Licia Sbattella

aa 2007-08

Translated and adapted by L. Breveglieri

DETERMINISTIC BOTTOM-UP SYNTAX ANALYSIS

The LL(1) method is not applicable if a state of a machine has two outgoing arcs labeled with two non-disjoint lookahead sets.

A technique for trying to circumvent this potential problem is that of deferring any decision that is presently undetermined due to insufficient information, going on with the computation and keeping still open all the possible computations until new evidence is obtained, which allows to make the deferred decision.

In the time interval from when uncertainlty appears to when it is eventually solved, the computation must store any intermediate information that is necessary to make the decision later.

Such a strategy is the basis for the bottom-up syntax analysis algorithms or LR(k) (L = left, the input string is scanned from left to right, R = right, the syntax tree is reconstructed starting from the leaves to the root or, equivalently, the derivation is reconstructed starting from the end to the beginning, $k \geq 1$ length of the lookahead window).

First the LR(0) method, then the LR(1) method; the latter allows to recognise all the deterministic free languages.

SHIFT AND REDUCTION ANALYSIS

Bottom-up syntax analyzers are a special subclass of a larger class of pushdown automata, which are called *shift-and-reduction* analyzers.

The bottom-up analyzer scans the input characters from left to right and pushes them orderly onto the stack, possibly encoding them into a suited stack alphabet (so-called *shift move*), until a top portion of the stack (so-called *reducible part*) happens to be the right side of a grammar rule; the reducible part is then popped and the left part of the corresponding rule is pushed, to replace the reducible part (so-called *reduction move*).

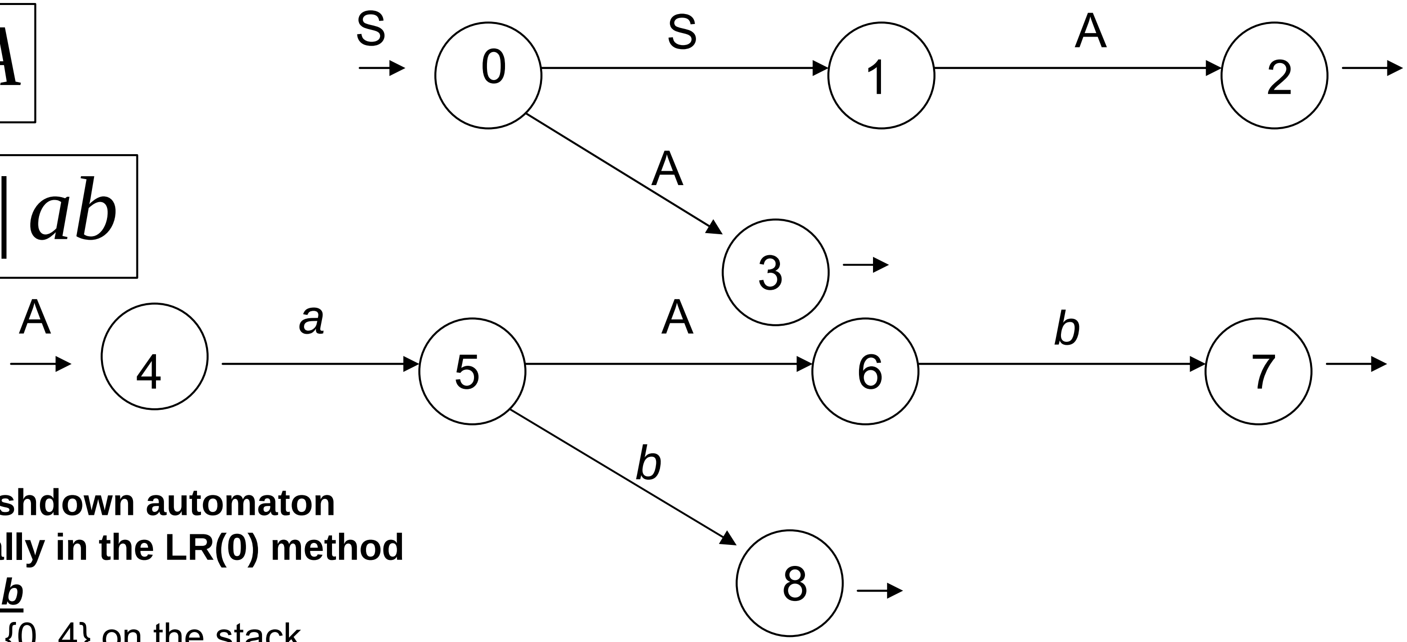
This process goes on with a new series of shift moves, until a new reducible part appers on the top of the stack, and then it is reduced, and so on alternating shift and reduction moves ...

The process terminates either when it is impossible to proceed any longer, as there are not any shift or reduction moves still enabled and in this case the string is rejected by error, or when the entire input string has been scanned and the stack contains only the grammar axiom, and then the input string is accepted.

EXAMPLE – language: non-empty list of constructs $a^n b^n$, $n \geq 1$; condition LL(k) fails for k = 0 (the grammar is left recursive and both S and A begin by a)

$$\boxed{S \to SA \mid A}$$

$$\boxed{A \to aAb \mid ab}$$



Computation of the pushdown automaton working deterministically in the LR(0) method

analysis of the string *ab*

- initial stack symbol I_0 = {0, 4} on the stack
- read *a* in the input, shift input head and go to state δ(I_0, *a*) = δ(0, *a*) U δ(4, *a*) = Ø U {5} = {5}
- complete {5} adding state 4 to it, as from 5 there is an outgoing arc labeled by A, thus forming a stack symbol I_2 = {4, 5}, then push I_2 onto the stack
- read *b* in the input, shift input head and go to state δ(I_2, *b*) = δ(4, *b*) U δ(5, *b*) = Ø U {8} = {8} = I_3, then push I_3 on the stack
- as state 8 is final, use rule A → *ab* to replace the reducible part on the stack by the non-terminal A
- such a reduction operation cancels from the top of the stack the part I_2 I_3 (a number of stack symbols equal to |*ab*| = 2, that is equal to the length of the right side of the grammar rule recognised by state 8)
- now the stack contains I_0 = {0, 4}, the automaton executes the move A (see that state 0 has an arc labeled by A going to state 3) and pushes I_4 onto the stack, where I_4 = δ(4, A) U δ (0, A) = Ø U {3} = {3}

Now, as 3 is a final state of the axiomatic machine, the pushdown automaton terminates and accepts.

LR(0) SYNTAX ANALYSIS – lookahead length = 0: the choice of the arc to take, that is the move, depends only on the current state of the currently active machine.

Construct a finite automaton, the so called LR(0) DRIVER MACHINE: this machine will drive the parser (which is a pushdown automaton) to recognize valid prefixes.

Differently from L(*k*) analysis, two or more machines (of the recursive network representing the grammar) may be simultaneously active, due to that LR analysis can deal with situations of uncertaintly. All the simultaneously active machines use the same stack (this resembles a cartesian product of finite state automata, though the construction has some differences).

The DRIVER GRAPH is made of MACROSTATES, which are sets of states of the machines of the recursive network (some of the component states may be final):

1. A final state included in a macrostate is said to be a REDUCTION STATE.
2. An ordinary state included in a macrostate is said to be a SHIFT STATE if it has an arc going to another state in the machine it belongs to.
3. A macrostate containing only reduction states is said to be REDUCTION.
4. A macrostate containing only shift states is said to be SHIFT.
5. Otherwise, a macrostate is said to be MIXED.

LR(0) CONDITION
A grammar satisfies the LR(0) condition if every macrostate of the driver graph satisfies both the following conditions:

1) the macrostate is not mixed

and

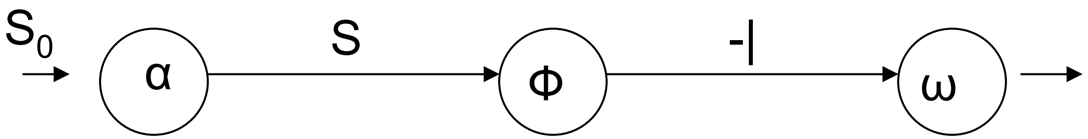
2) the macrostate contains exactly one reduction state (that is, it does not contain other reduction states or any shift state)

The family of LR(0) LANGUAGES is the set of all the free languages that are generated by grammars that satisfy the LR(0) condition.

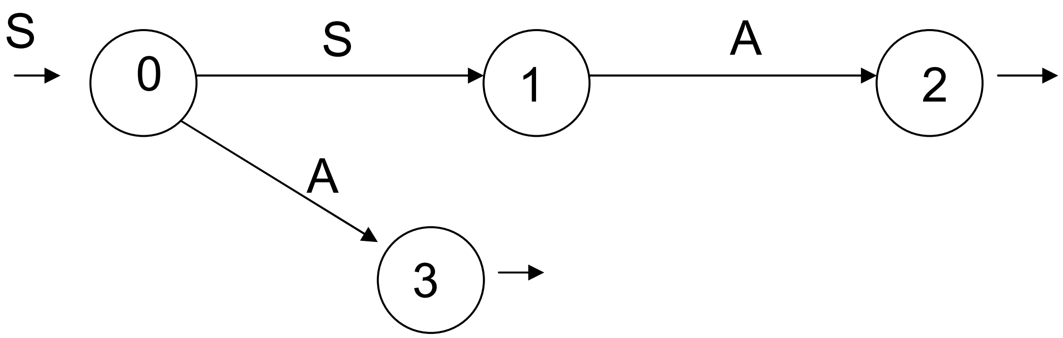
If a grammar G is of the LR(0) type, the driver graph can be easily used to construct the complete deterministic pushdown automaton that recognises the phrases of the language L(G) and constructs the corresponding syntax trees. The driver graph itself is a sort of condensed form of the pushdown automaton and contains only essential information; the pushdown automaton needs some more detail, easy to add.

EXAMPLE – construction of the driver graph – recognizer of valid prefixes – the language is that of the non-empty lists of constructs of the type $a^n b^n$, $n \geq 1$.

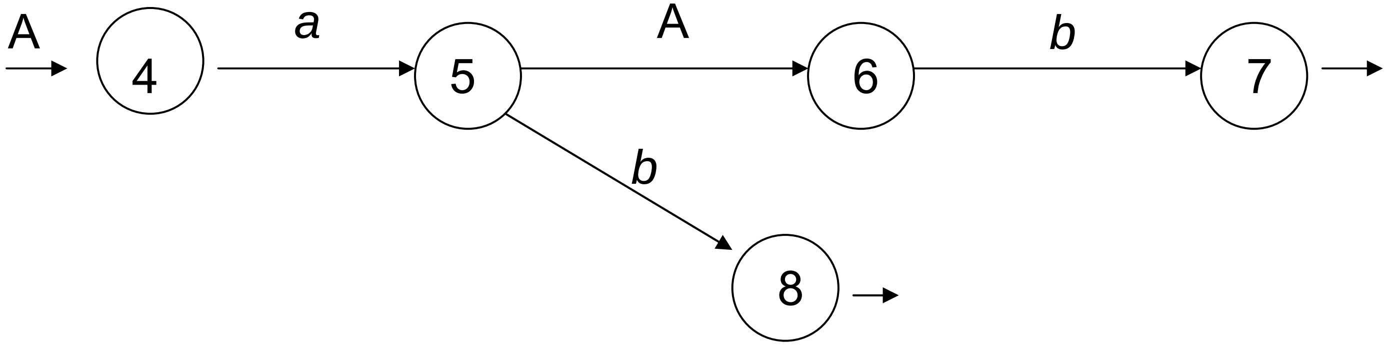
$$\boxed{S_0 \to S\text{-}|}$$



$$\boxed{S \to SA \mid A}$$



$$\boxed{A \to aAb \mid ab}$$



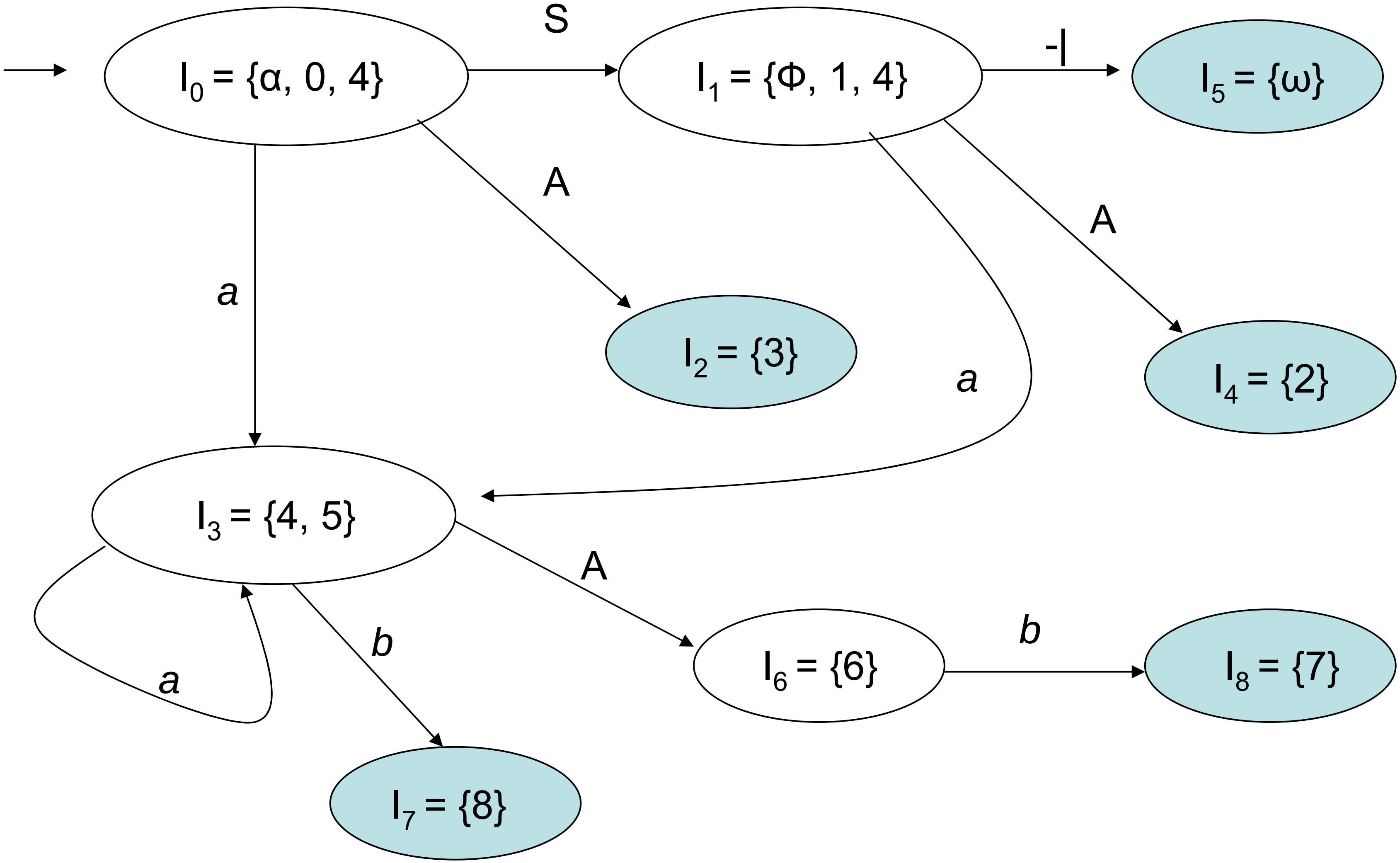
How to compute the macrostates and the arcs (or moves) of the driver graph

macrostates	construction of the macrostates	arc (or move)
0. $\varnothing$	$close(\alpha) = \{\alpha, 0, 4\} = I_0$	
1. $I_0 = \{\alpha, 0, 4\}$	$close\ (\delta(\alpha, S) \cup \delta(0, S) \cup \delta(4, S)) =$	$\vartheta(I_0, S) = I_1$
	$close\ (\{\phi, 1\}) = \{\phi, 1, 4\} = I_1$	
2. $I_0 = \{\alpha, 0, 4\}, I_1$	$close\ (\delta(\alpha, A) \cup \delta(0, A) \cup \delta(4, A)) =$	$\vartheta(I_0, A) = I_2$
	$close\ (3) = \{3\} = I_2$	
3. $I_0 = \{\alpha, 0, 4\}, I_1, I_2$	$close\ (\delta(\alpha, a) \cup \delta(0, a) \cup \delta(4, a)) =$	$\vartheta(I_0, a) = I_3$
	$close\ (5) = \{4, 5\} = I_3$	
4. $I_0, I_1 = \{\phi, 1, 4\}, I_2, I_3$	$close\ (\delta(\phi, A) \cup \delta(1, A) \cup \delta(4, A)) =$	$\vartheta(I_1, A) = I_4$
	$close\ (2) = \{2\} = I_4$	
5. $I_0, I_1 = \{\phi, 1, 4\}, I_2, I_3, I_4$	$close\ (\delta(\phi, a) \cup \delta(1, a) \cup \delta(4, a)) =$	$\vartheta(I_1, a) = I_3$
	$close\ (5) = \{4, 5\} = I_3$	

chius indicates the closure operation

How to compute the macrostates and the arcs (continued)

macrostates	construction of the macrostates	arc (or move)
6. $I_0, I_1 = \{\phi, 1, 4\}, I_2, I_3, I_4$	$close\ (\delta(\phi, \dashv) \cup \delta(1, \dashv) \cup \delta(4, \dashv)) =$	$\vartheta(I_1, \dashv) = I_5$
	$close\ (\omega) = \{\omega\} = I_5$	
7. $I_0, I_1, I_2, I_3 = \{4, 5\}, I_4, I_5$	$close\ (\delta(4, A) \cup \delta(5, A)) =$	$\vartheta(I_3, A) = I_6$
	$close\ (6) = \{6\} = I_6$	
8. $I_0, I_1, I_2, I_3 = \{4, 5\}, I_4, I_5, I_6$	$close\ (\delta(4, a) \cup \delta(5, a)) =$	$\vartheta(I_3, a) = I_3$
	$close\ (5) = \{4, 5\} = I_3$	
9. $I_0, I_1, I_2, I_3 = \{4, 5\}, I_4, I_5, I_6$	$close\ (\delta(4, b) \cup \delta(5, b)) =$	$\vartheta(I_3, b) = I_8$
	$close\ (8) = \{8\} = I_7$	
10. $I_0, I_1, I_2, I_3, I_4, I_5, I_6 = \{6\}, I_7$	$close\ (\delta(6, b)) =$	$\vartheta(I_6, b) = I_8$
	$close\ (7) = \{7\} = I_8$	
11. $I_0, I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8$		alt


I0 = {α, 0, 4}
S
I1 = {Φ, 1, 4}
-|
I5 = {ω}
A
A
a
I2 = {3}
a
I4 = {2}
I3 = {4, 5}
A
b
b
I6 = {6}
I8 = {7}
a
I7 = {8}

PROPERTIES OF THE LR(0) DRIVER GRAPH

1. All the arcs ingoing into a macrostates are labeled in the same way.
2. Every reduction macrostate does not have any successor.
3. Every shift (or mixed) macrostate has at least one successor.

What is the language recognised by the driver graph (conceived as a finite state automaton where every state is final) ? In the previous example the strings are:

S, A, *a*, S-|, SA, *a*A, *aa*, *ab*, *a*Ab, *aaa*, *aab*, *aa*A*b*, ...

All these strings are prefixes of valid sentential forms generated by grammar G, suppposing the derivation order is rightmost. For instance, the following rightmost derivation:

$$S_0 \Rightarrow S\text{-}| \Rightarrow A\text{-}| \Rightarrow aAb\text{-}| \Rightarrow aaAbb\text{-}| \Rightarrow aaabbb\text{-}|$$

contains the following prefixes, accepted by the driver graph:

S, S-|, A, A-|, *a*A, *a*A*b*, *aa*A*b*, *aaab*

But not all the prefixes of the right sentential forms are accepted by the driver graph (for instance *aaabb* is not). The motivation is that the driver graph executes some computations in parallel, until one of them reaches a reduction macrostate.

SHIFT-AND-REDUCTION ANALYZER

If a grammar is LR(0), the driver graph allows to construct easily the deterministic pushdown automaton that recognizes the language L(G) and builds the syntax trees.

The LR(0) analyzer (which is a pushdown automaton) works as follows:

1) Each macrostate is associated with the common label of the ingoing arcs, and the stack symbols are the macrostate names themselves; possibly associate to each name the ingoing label, for better clarity (this is not necessary anyway).
2) The analyzer examines the current input character and, if the stack top symbol is I, executes the move prescribed by the driver graph in macrostate I, and pushes the next macrostate (this is the shift move)
3) If the current macrostate (that on the top of the stack) is of the reduction type (and hence contains exctly one reduction state), the analyzer executes a series of operations called reduction. Such operations simulate one derivation step of the grammar and change the contents of the stack (this is the reduction move).
4) At the end of the input string, the analyzer accepts the string if the stack is empty or contains only the initial macrostate, otherwise rejects (acceptance/reject move).

RIGHTMOST DERIVATION SIMULATED IN REVERSE ORDER (from the end to the beginning)

stack	x	comment
I_0	a a b b a b -|	shift
$I_0\ aI_3$	a b b a b -|	shift
$I_0\ aI_3\ aI_3$	b b a b -|	shift
$I_0\ aI_3\ aI_3\ bI_7$	b a b -|	reduce by $A \to ab$
$I_0\ aI_3\ AI_6$	b a b -|	shift
$I_0\ aI_3\ AI_6\ bI_8$	a b -|	reduce by $A \to aAb$
$I_0\ AI_2$	a b -|	reduce by $S \to A$
$I_0\ SI_1$	a b -|	shift
$I_0\ SI_1\ aI_3$	b -|	shift
$I_0\ SI_1\ aI_3\ bI_7$	-|	reduce by $A \to ab$
$I_0\ SI_1\ AI_4$	-|	reduce by $\mathrm{S} \to SA$
$I_0\ SI_1$	-|	shift
$I_0\ SI_1$ -|I_5		reduce by $S_0 \to S$-|
I_0		accept

order of the reduction moves

$$A \to ab, A \to aAb, S \to A, A \to ab, S \to SA, S_0 \to S\text{-}|$$

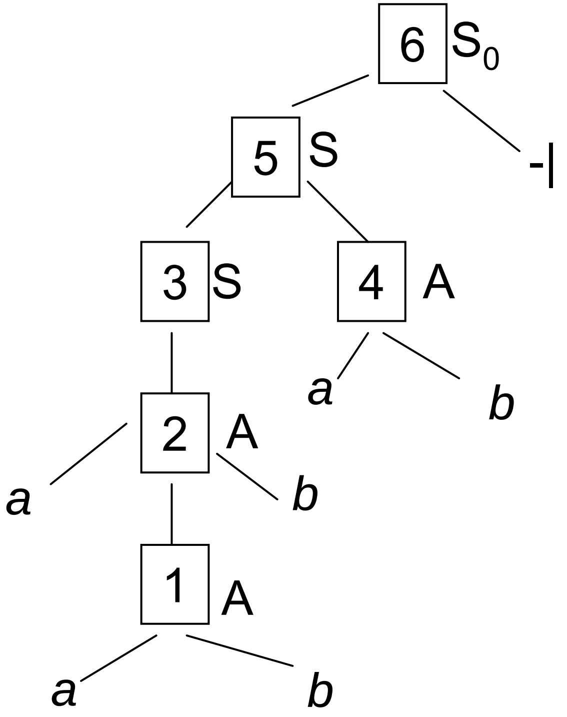
chain of the corresponding reductions

$$aabbab\text{-}| \Rightarrow aAbab\text{-}| \to Aab\text{-}| \Rightarrow Sab\text{-}| \Rightarrow SA\text{-}| \Rightarrow S\text{-}| \Rightarrow S_0$$

1 2 3 4 5 6

The parser is of the bottom-up type: the derivation is reconstructed in reverse order, from the end to the beginning.

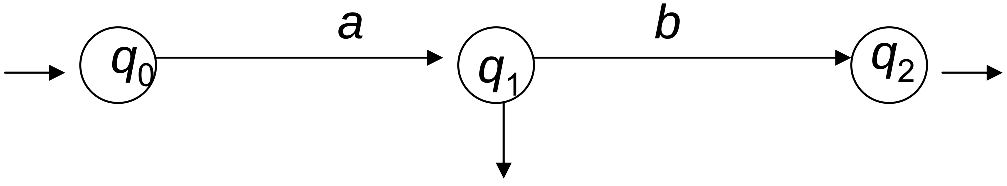
$$\boxed{S_0 \overset{+}{\Rightarrow} aabbab\text{-}|}$$

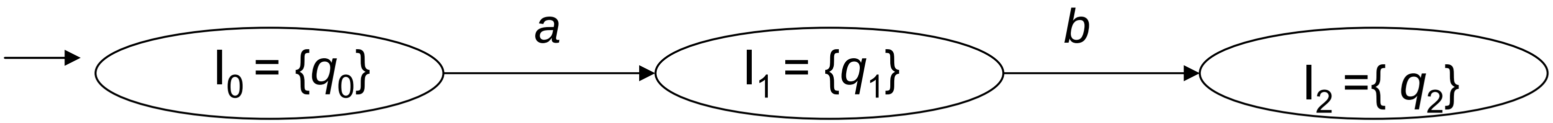


A language containing phrases that are prefixes of other language phrases, is certainly not LR(0).

EXAMPLE - L = {*a*, *ab*}

$$S \to a \mid ab$$



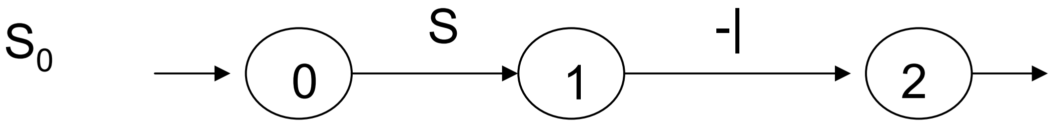


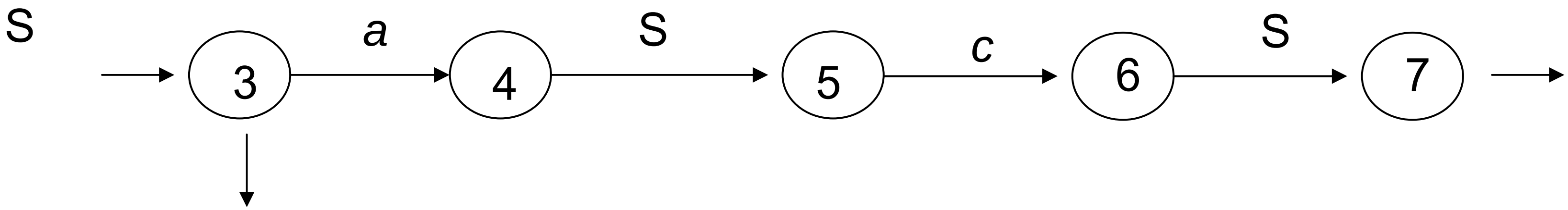
A grammar containing null rules certainly violates the LR(0) condition. In fact, a rule of the type A→ ε | β causes the initial state $q_{A,0}$ of machine A to be final as well. But such a state also has an outgoing arc leading to the recognition of the (non-empty) right part β. As a consequence, the macrostate containing state $q_{A,0}$ violates condition LR(0).

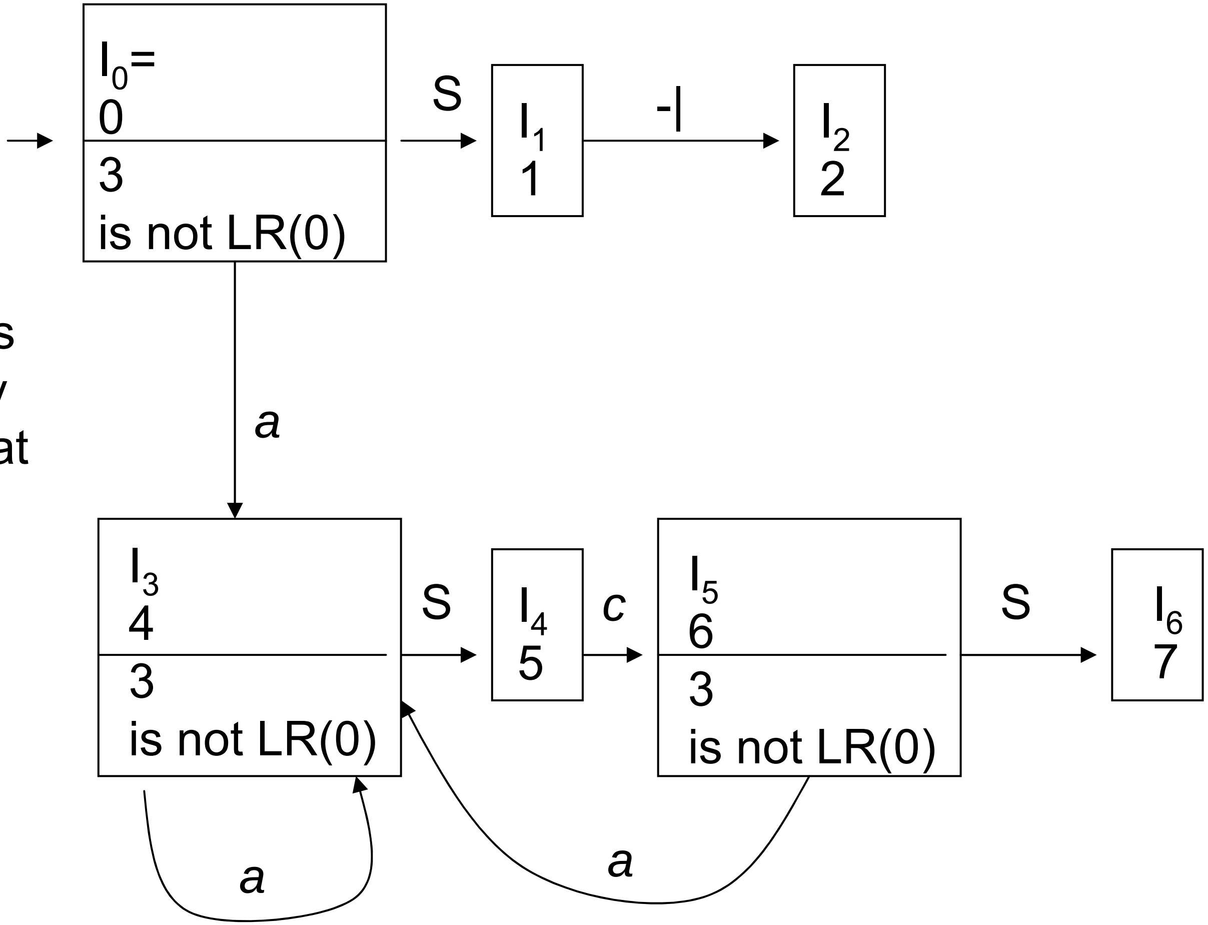
EXAMPLE - the grammar below of the Dyck language is not LR(0)

$$S_0 \to S\text{-|}$$

$$S \to aScS \mid \varepsilon$$







In the graph: every macrostate is divided horizontally into the subsets that are computed by the closure (or completion) process.

All the macrostates containing state 3 are mixed, and hence inadequate for LR(0) analysis.

Adding lookahead to the driver graph

To enhance the LR(0) method and obtain the LR(1) method, add to each macrostate some lookahead information.

As every macrostate is a set of states, it is necessary to compute the lookahead set of the terminal characters for each state and for each outgoing arc.

Intuitively, a macrostate is adequate for the LR(1) analysis if the next move of the analyzer (the pushdown automaton) is determined in a unique way by the macrostate currently on the top of the stack and by the lookahead set containing the current input character.

The LR(1) driver graph is constructed similarly to the LR(0) case, but every macrostate I of the LR(1) driver graph is a set of pairs, called *candidates*, like: $<q, a> \in Q \times (\Sigma \cup \{\dashv\})$, where Q is the set of the states of the machine network. Two or more candidates sharing the same state are grouped, for brevity:

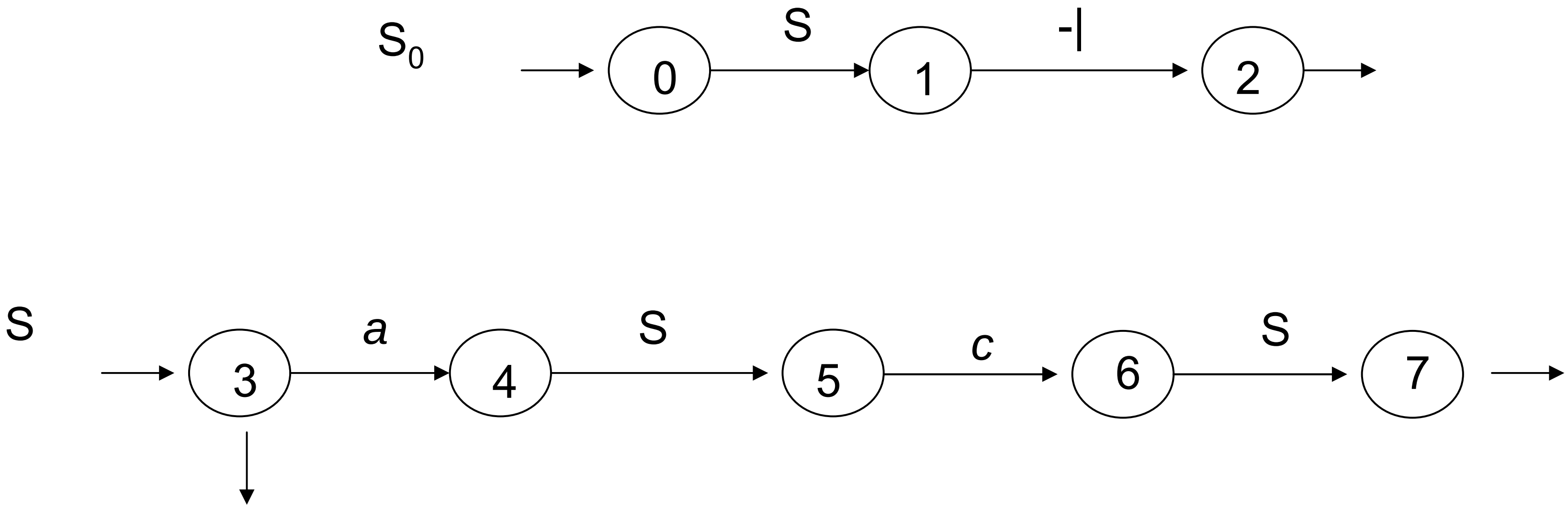
$$<q, \{a_1, a_2, \ldots, a_k\}> \equiv \{<q, a_1>,<q, a_2>, \ldots <q, a_k>\}$$

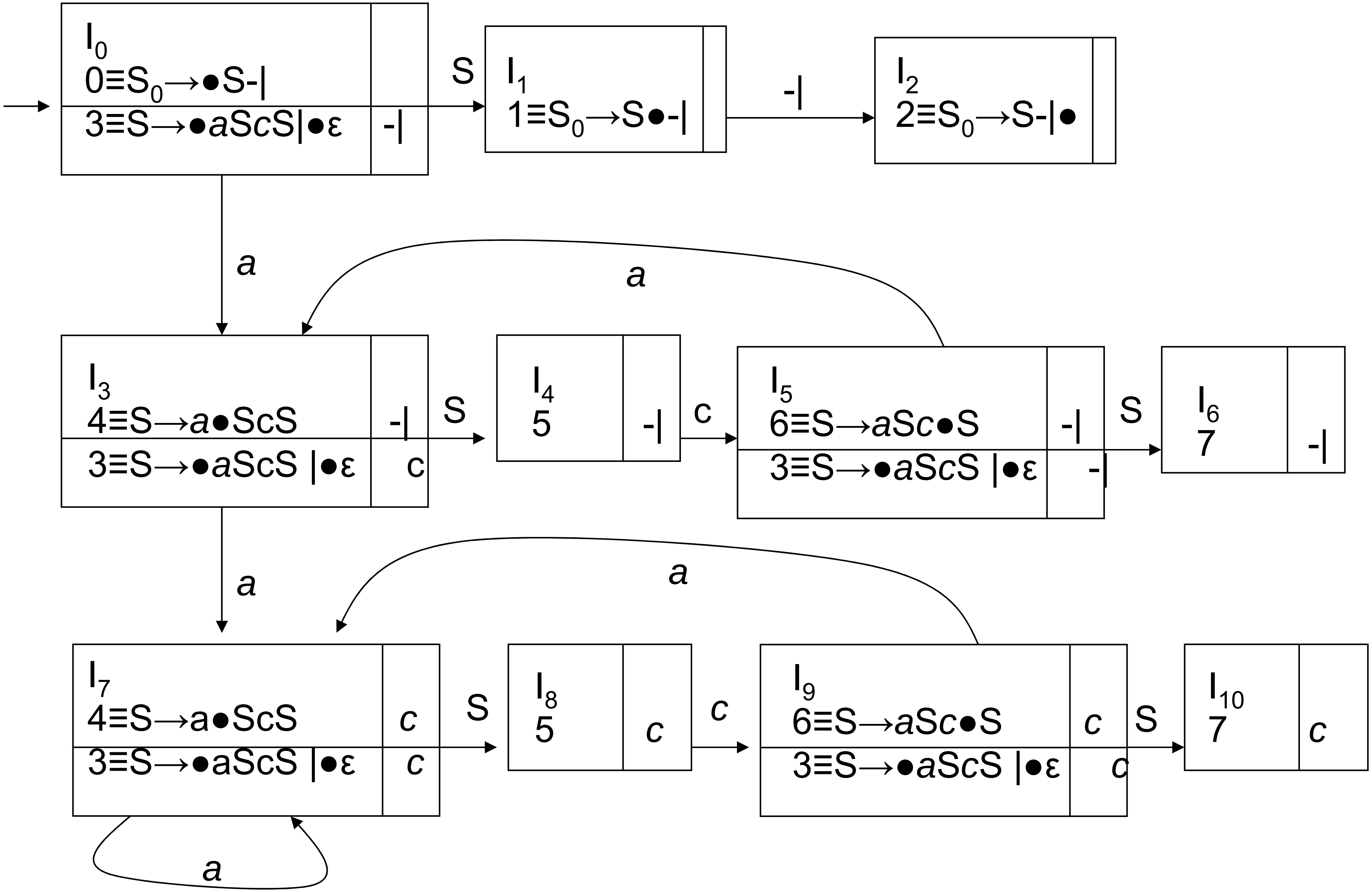
The set $\{a_1, a_2, \ldots, a_k\}$ is the *lookahead set* of the state q.

EXAMPLE – the grammar below of the Dyck langauge is LR(1) – continued

$S_0 \to S$-|

$S \to aScS \mid \varepsilon$





I_0
$0 \equiv S_0 \rightarrow \bullet S\text{-}|$
$3 \equiv S \rightarrow \bullet aScS | \bullet \varepsilon$ -|

I_1
$1 \equiv S_0 \rightarrow S \bullet \text{-}|$

I_2
$2 \equiv S_0 \rightarrow S\text{-}| \bullet$

I_3
$4 \equiv S \rightarrow a \bullet ScS$ -|
$3 \equiv S \rightarrow \bullet aScS | \bullet \varepsilon$ c

I_4
5   -|

I_5
$6 \equiv S \rightarrow aSc \bullet S$ -|
$3 \equiv S \rightarrow \bullet aScS | \bullet \varepsilon$ -|

I_6
7   -|

I_7
$4 \equiv S \rightarrow a \bullet ScS$ c
$3 \equiv S \rightarrow \bullet aScS | \bullet \varepsilon$ c

I_8
5   c

I_9
$6 \equiv S \rightarrow aSc \bullet S$ c
$3 \equiv S \rightarrow \bullet aScS | \bullet \varepsilon$ c

I_{10}
7   c

LR(1) CONDITION

Macrostates can still be classified as in the LR(0) case ($k = 0$):

1) A candidate is REDUCTION if it contains a final state of one of the machines of the network of machines (such a state is associated with a completed grammar rule).
2) A candidate is said to be SHIFT if it contains a state with an outgoing arc (labeled with a terminal or non-terminal symbol).

However, the LR(1) condition is less drastic than the LR(0) condition:

LR(1) CONDITION – a grammar satisfies the LR(1) condition if for every macrostate both the following conditions hold:

1. there is not any conflict between reduction and shift: every reduction candidate has a lookahead set disjoint from the terminal labeling the arcs outgoing from the macrostate the reduction candidate belongs to

and

2. there is not any reduction-reduction conflict: any two reduction candidates in the same macrostate have disjoint lookahead sets

PREVIOUS EXAMPLE – how to check condition LR(1)

Macrostates I_1, I_2, I_4, I_6, I_8 and I_{10} contain only one candidate each, and hence they satisfy the LR(1) condition immediately.

The remaining macrostates contain the reduction candidate S→ ● ε (equivalent to S →ε ●, as the right side is null), the shift candidate S → ● *a* S *c* S – both associated with state 3 – and another shift candidate.

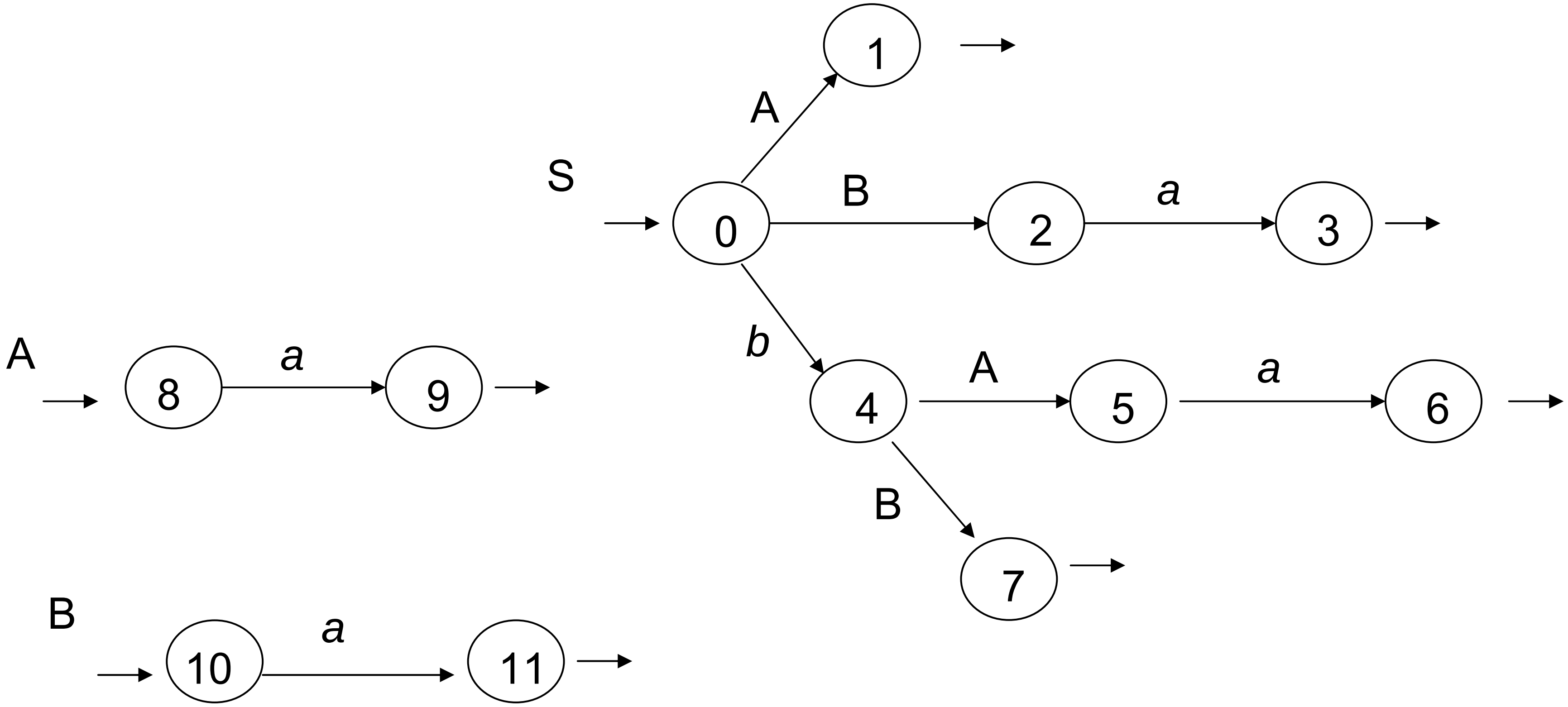
Disjunction check:

The grammar proves to be LR(1).

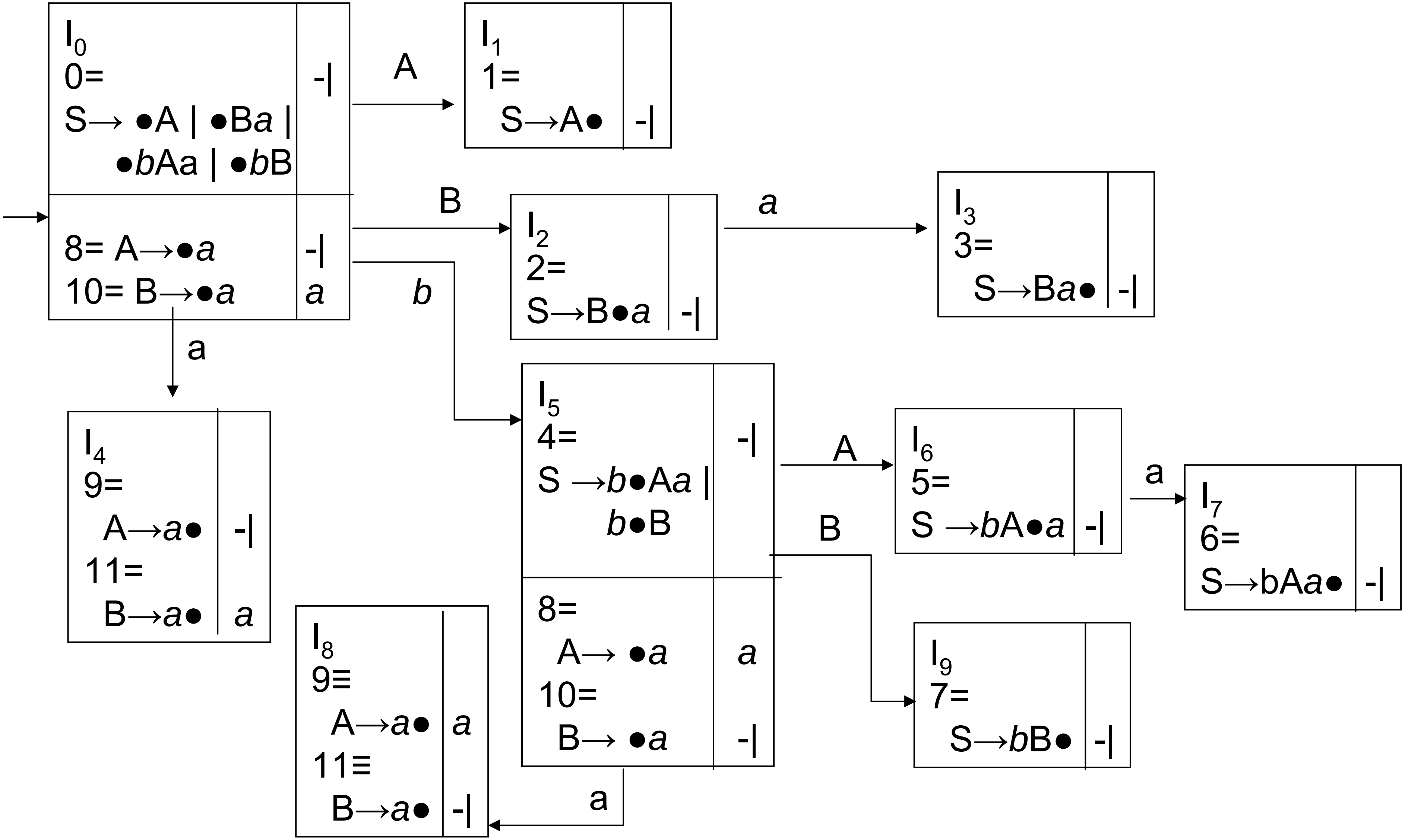
macrostate	lookahead	outgoing label	condition
I_0	-|	a	$\{\text{-\|}\} \cap \{a\} = \emptyset$
I_3	c	a	$\{c\} \cap \{a\} = \emptyset$
I_5	-|	a	$\{\text{-\|}\} \cap \{a\} = \emptyset$
I_7	c	a	$\{c\} \cap \{a\} = \emptyset$
I_9	c	a	$\{c\} \cap \{a\} = \emptyset$

EXAMPLE - LR(1) check with a macrostate containing two reduction candidates

$$S \rightarrow A \mid Ba \mid bAa \mid bB \qquad A \rightarrow a \qquad B \rightarrow a$$



LR(1) driver graph where reduction candidates are highlighted.



I_0
0= -|
S→ •A | •B*a* |
•*b*Aa | •*b*B
8= A→•*a* -|
10= B→•*a* *a*

I_1
1=
S→A• -|

I_2
2=
S→B•*a* -|

I_3
3=
S→B*a*• -|

I_4
9=
A→*a*• -|
11=
B→*a*• *a*

I_5
4= -|
S →*b*•A*a* |
b•B
8=
A→ •*a* *a*
10=
B→ •*a* -|

I_6
5=
S →*b*A•*a* -|

I_7
6=
S→bA*a*• -|

I_8
9≡
A→*a*• *a*
11≡
B→*a*• -|

I_9
7=
S→*b*B• -|

In the previous figure the macrostates that could violate the LR(1) condition are I_4 and I_8. All the others already satisfy the condition LR(0) (and hence also LR(1)).

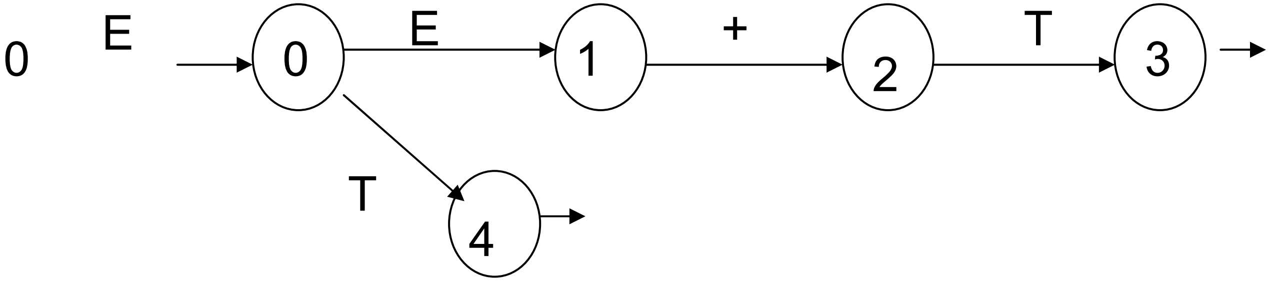
Each of the two macrostates mentioned above contains two reduction candidates, but their lookahead sets are disjoint and therefore part 2 of the above LR(1) condition is satisfied. Moreover, from none of the two macrostates there are outgoing arcs, hence part 1 of the previous LR(1) condition is also satisfied.

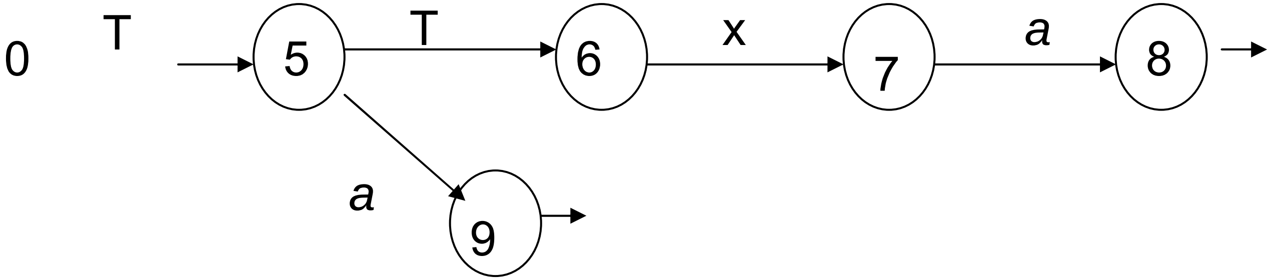
The grammar is LR(1), but not LR(1), as in I_4 and I_8 there are two reductions, between which only lookahead allows to select in a deterministic way.

EXAMPLE – arithmetic expressions – LR(1) condition and the role of lookahead sets

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T \times a \mid a$$



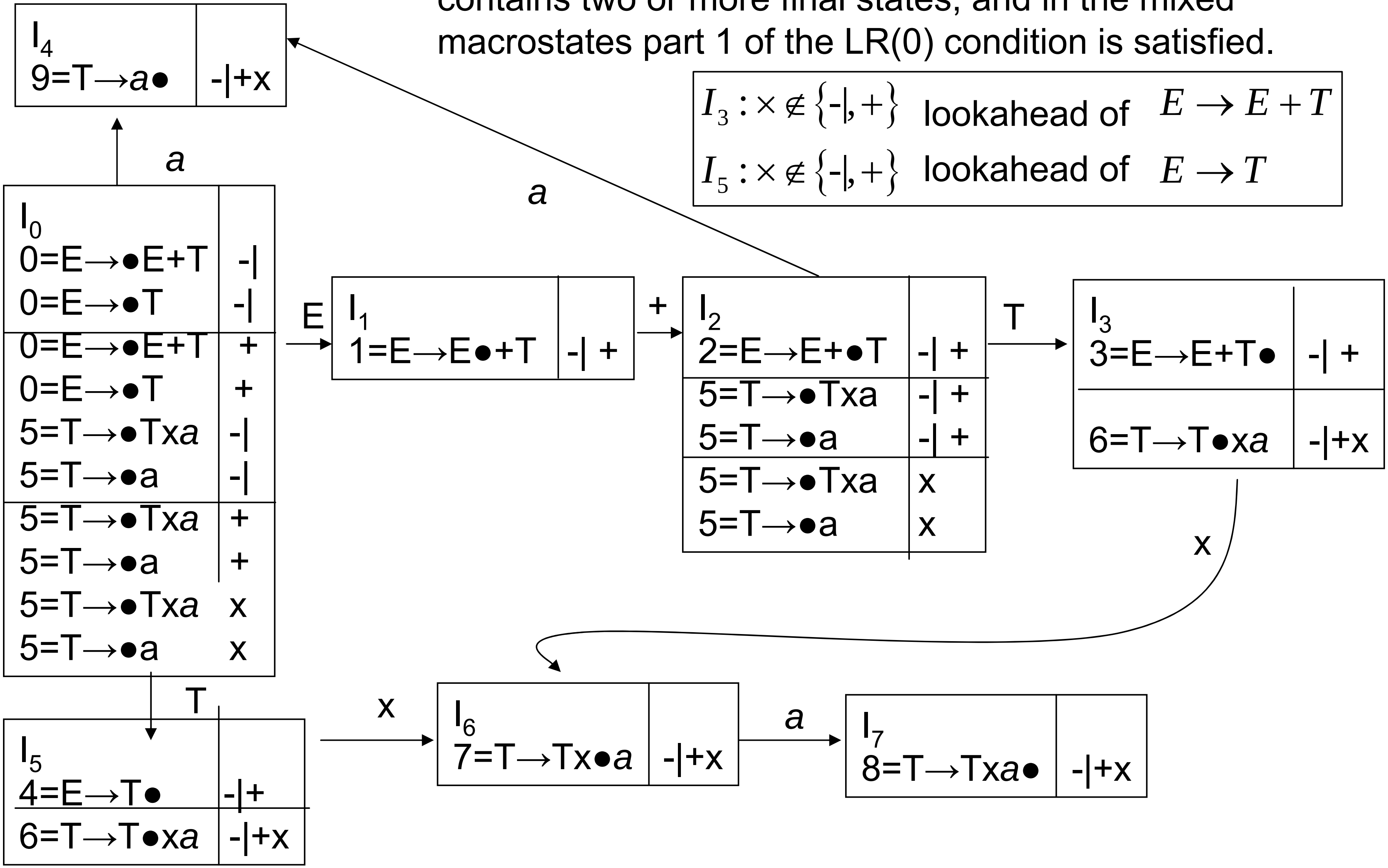


LR(1) driver graph

The grammar proves to be LR(1). In fact no macrostate contains two or more final states, and in the mixed macrostates part 1 of the LR(0) condition is satisfied.

$I_3 : \times \notin \{-|, +\}$ lookahead of $E \to E+T$

$I_5 : \times \notin \{-|, +\}$ lookahead of $E \to T$



I_0	
0=E→●E+T	-|
0=E→●T	-|
0=E→●E+T	+
0=E→●T	+
5=T→●Tx*a*	-|
5=T→●a	-|
5=T→●Tx*a*	+
5=T→●a	+
5=T→●Tx*a*	x
5=T→●a	x

I_1	
1=E→E●+T	-| +

I_2	
2=E→E+●T	-| +
5=T→●Txa	-| +
5=T→●a	-| +
5=T→●Txa	x
5=T→●a	x

I_3	
3=E→E+T●	-| +
6=T→T●x*a*	-|+x

I_4	
9=T→*a*●	-|+x

I_5	
4=E→T●	-|+
6=T→T●x*a*	-|+x

I_6	
7=T→Tx●*a*	-|+x

I_7	
8=T→Tx*a*●	-|+x

LALR(1) – is a determinism condition intermediate between LR(0) and LR(1)

Simplify the LR(1) driver graph, folding those macrostates that are indistinguishable in the LR(0) driver graph. In the practice, fold those macrostates that contain candidates differing only as for the lookahead sets. However, retain the lookahead sets (simply unite the sets).

The LALR(1) driver graph is isomorphic to the LR(1) drievr graph, but may contain mixed macrostates or reduction macrostates with two or more reduction candidates.

A grammar is said to be LALR(1) if both conditions below hold true:

1. there is not any conflict between reduction and shift: every reduction candidate has a lookahead set disjoint from the terminal labeling the arcs outgoing from the macrostate

and

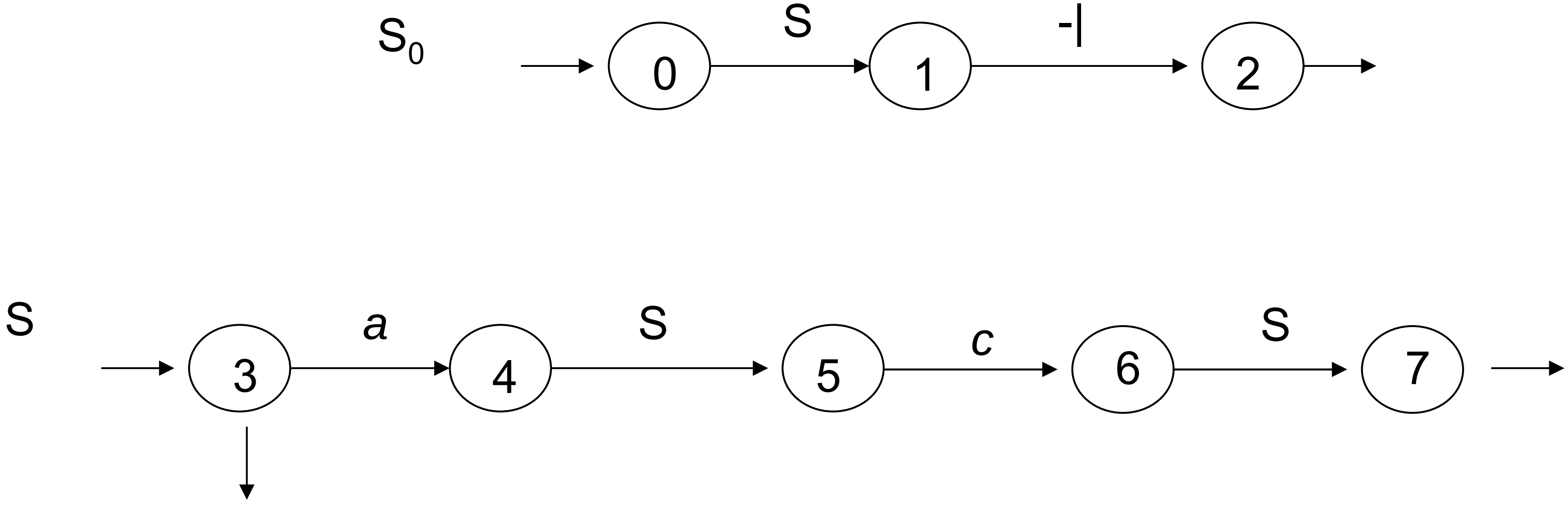
2. there is not any reduction-reduction conflict: any two reduction candidates in the same macrostate have disjoint lookahead sets

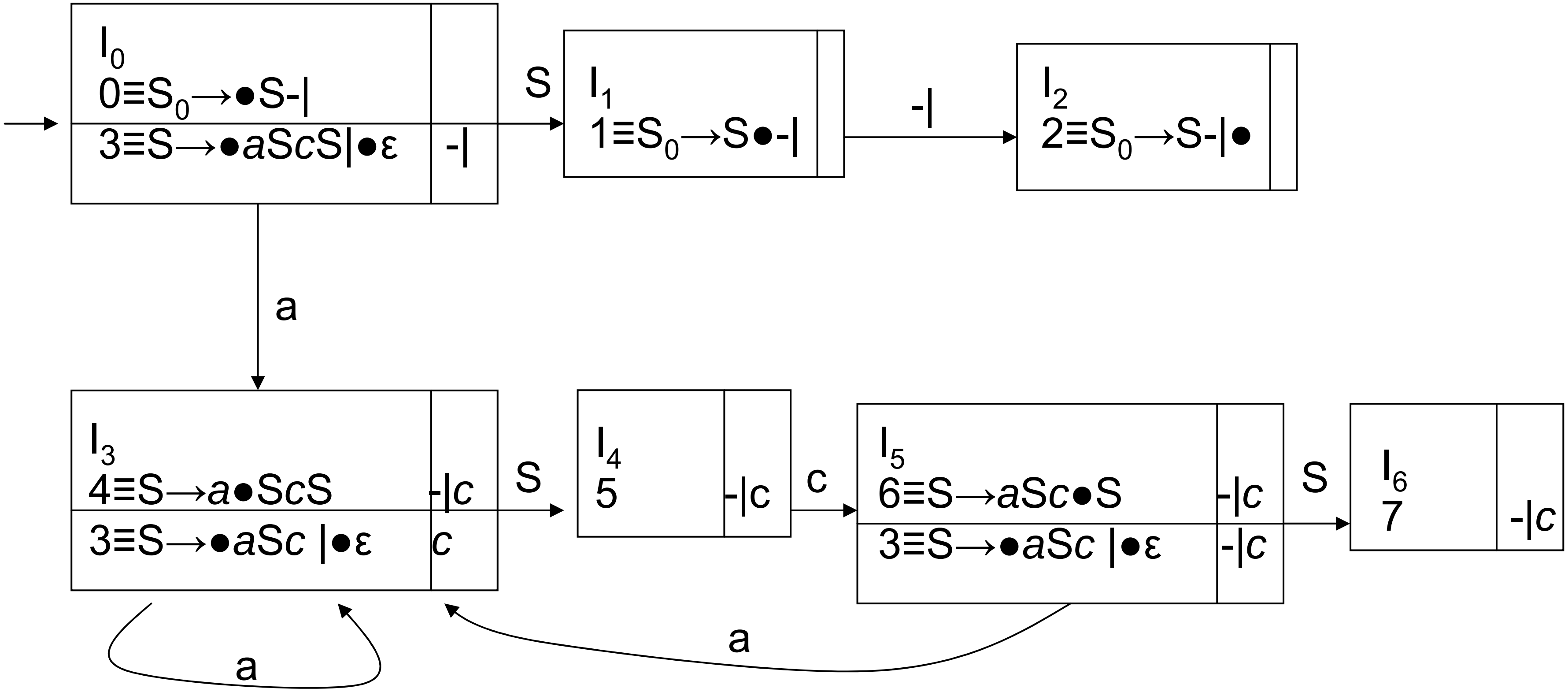
(note: the LALR(1) condition is identical to the LR(1) condition)

The family of LALR(1) grammars is strictly contained in the family of LR(1) grammars, and contains strictly the family of LR(0) grammars.

EXAMPLE – the Dyck grammar is LALR(1)

$$S_0 \to S\text{-|}$$
$$S \to aScS \mid \varepsilon$$




I_0
$0 \equiv S_0 \to \bullet S\text{-}|$
$3 \equiv S \to \bullet aScS \mid \bullet \varepsilon$
-|
S
I_1
$1 \equiv S_0 \to S \bullet \text{-}|$
-|
I_2
$2 \equiv S_0 \to S\text{-}| \bullet$
a
I_3
$4 \equiv S \to a \bullet ScS$
-|c
$3 \equiv S \to \bullet aSc \mid \bullet \varepsilon$
c
S
I_4
5
-|c
c
I_5
$6 \equiv S \to aSc \bullet S$
-|c
$3 \equiv S \to \bullet aSc \mid \bullet \varepsilon$
-|c
S
I_6
7
-|c
a
a

No macrostate contains two or more reduction candidates. In the mixed macrostates, the lookahead of the reduction candidate S → ε is disjoint from the outgoing labels. Note that the LALR(1) driver graph performs the same task as that of the LR(1) driver graph, but using a lower number of macrostates.

<u>Example of slide n. 26</u> : the LR(1) driver graph is isomorphic to the LR(0) one, as the macrostates are the same but for their respective lookahead sets. Grammar E → E + T | T T → T x *a* | *a* satisfies condition LALR(1).

<u>Example of slide n. 23</u> – the grammar is LR(1) but not LALR(1). Overlapping the two macrostates I_4 and I_8, which are indistinguishable without lookahead, one gets the macrostate of the LALR(1) driver graph:

<table>
<tr><td>I_4, I_8</td><td></td></tr>
<tr><td>9=A→<i>a</i>●</td><td>-|<i>a</i></td></tr>
<tr><td>11=B→<i>a</i>●</td><td>-|<i>a</i></td></tr>
</table>

But in this macrostate there are two reduction candidates with identical lookahead (hence non-disjoint), therefore such a macrostate is not LALR(1).

The LALR(1) method has often proved to be adequate to design practical compilers fro real programming languages. Sometimes it is necessary to modify slightly the original grammar, to make it LALR(1).

EXAMPLE – analysis of the string *a* + *a*
For arithmetic expressions with driver graph LR(1) .... or LALR(1)

stack	x	comment
I_0	$a+a$-|	shift
$I_0\ aI_4$	$+a$-|	reduce $T \to a$
$I_0\ TI_5$	$+a$-|	$+ \in$ prosp. of 4:
		reduce $E \to T$
$I_0\ EI_1$	$+a$-|	shift
$I_0\ EI_1\ +I_2$	a-|	shift
$I_0\ EI_1\ +I_2\ aI_4$	-|	reduce $T \to a$
$I_0\ EI_1\ +I_2\ TI_3$	-|	reduce $E \to E+T$
I_0	-|	accept

Bibliography

- – S. Crespi Reghizzi, *Linguaggi Formali e Compilazione*, Pitagora Editrice Bologna, 2006
- – Hopcroft, Ullman, *Formal Languages and their Relation to Automata*, Addison Wesley, 1969
- – A. Salomaa – *Formal Languages*, Academic Press, 1973
- – D. Mandrioli, C. Ghezzi – *Theoretical Foundations of Computer Science*, John Wiley & Sons, 1987
- – L. Breveglieri, S. Crespi Reghizzi, *Linguaggi Formali e Compilatori: Temi d'Esame Risolti,* web site (eng + ita)