FORMAL LANGUAGES AND COMPILERS

prof.s Luca Breveglieri and Angelo Morzenti

Exam of Wed 30 JANUARY 2019 - Part Theory

WITH SOLUTIONS - FOR TEACHING PURPOSES HERE THE SOLUTIONS ARE WIDELY COMMENTED

LAST + FIRST NAME:

(capital letters please)

MATRICOLA: SIGNATURE:

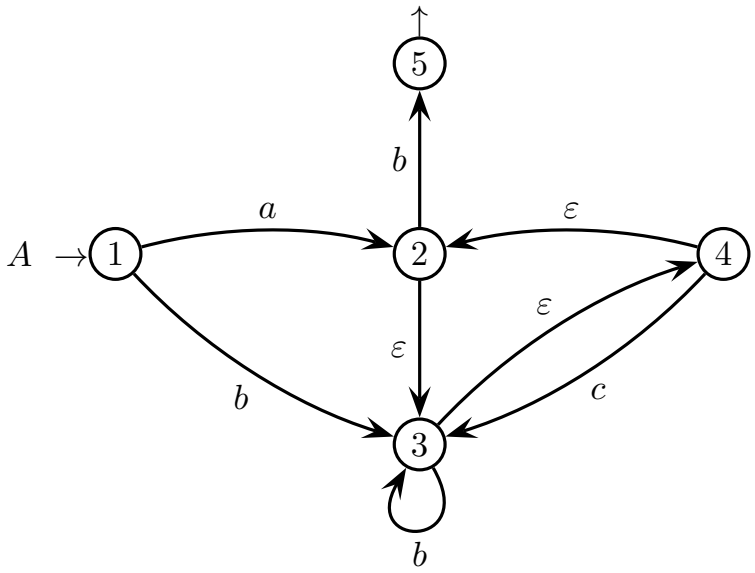
(or PERSON CODE)

INSTRUCTIONS - READ CAREFULLY:

- The exam is in written form and consists of two parts:
  1. Theory (80%): Syntax and Semantics of Languages
     - regular expressions and finite automata
     - free grammars and pushdown automata
     - syntax analysis and parsing methodologies
     - language translation and semantic analysis
  2. Lab (20%): Compiler Design by Flex and Bison
- To pass the exam, the candidate must succeed in both parts (theory and lab), in one call or more calls separately, but within one year (12 months) between the two parts.
- To pass part theory, the candidate must answer the mandatory (not optional) questions; notice that the full grade is achieved by answering the optional questions.
- The exam is open book: textbooks and personal notes are permitted.
- Please write in the free space left and if necessary continue on the back side of the sheet; do not attach new sheets and do not replace the existing ones.
- Time: part lab 60m - part theory 2h.15m

1 Regular Expressions and Finite Automata 20%

1. Consider the nondeterministic finite-state automaton A below, over a three-letter alphabet $\Sigma = \{ a, b, c \}$:



Answer the following questions:

(a) Examine automaton A and list all the accepted strings of length from 0 up to 3 (included). Say if automaton A is ambiguous and explain why yes or no.

(b) Cut all the spontaneous transitions (ε-transitions) of automaton A and obtain an equivalent automaton A', without spontaneous transitions, no matter if still nondeterministic.

(c) By using the Berry-Sethi (BS) method, obtain a deterministic automaton A'' equivalent to automaton A. You may choose to start from automaton A or A'. Say if automaton A'' is minimal (explain why yes or no) and minimize it, if necessary.

(d) Test that the automaton A'' obtained at point (c) is correct, by verifying it with the short strings found at point (a).

(e) (optional) Determine whether language $L(A)$ is local and explain why yes or no.

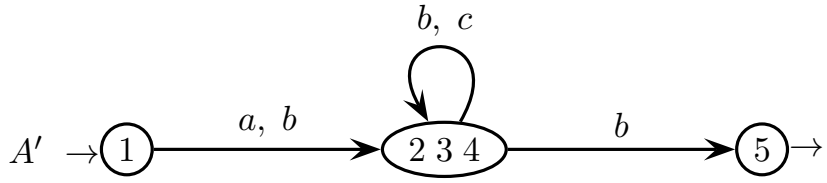
Solution

(a) Here are the short strings (length ≤ 3) recognized by automaton A (six in total):

$$a\,b \qquad b\,b \qquad a\,b\,b \qquad b\,b\,b \qquad a\,c\,b \qquad b\,c\,b$$

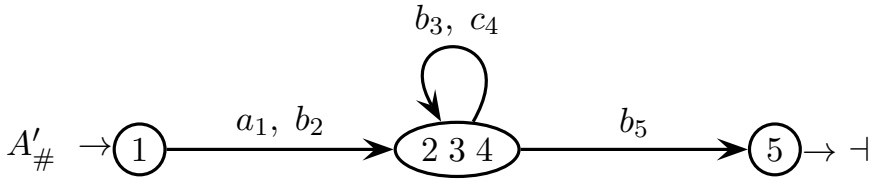
Automaton A is ambiguous: it has an ε-loop, namely $2 \xrightarrow{\varepsilon} 3 \xrightarrow{\varepsilon} 4 \xrightarrow{\varepsilon} 2$, which enables two or more accepting paths for the same string. For instance: $1 \xrightarrow{a} 2 \xrightarrow{b} 5$ and $1 \xrightarrow{a} 2 \xrightarrow{\varepsilon} 3 \xrightarrow{\varepsilon} 4 \xrightarrow{\varepsilon} 2 \xrightarrow{b} 5$, both for the valid string $a\,b$. All the valid strings can loop unlimitedly many times, thus they are infinitely ambiguous.

(b) It is well known that the states on an ε-loop can be merged, as when the automaton enters one such state, it can soon enter all the other states in the loop, without reading anything in the input. Thus, merge the states 2, 3 and 4 of the ε-loop $2 \xrightarrow{\varepsilon} 3 \xrightarrow{\varepsilon} 4 \xrightarrow{\varepsilon} 2$ of automaton A into one state (2 3 4):

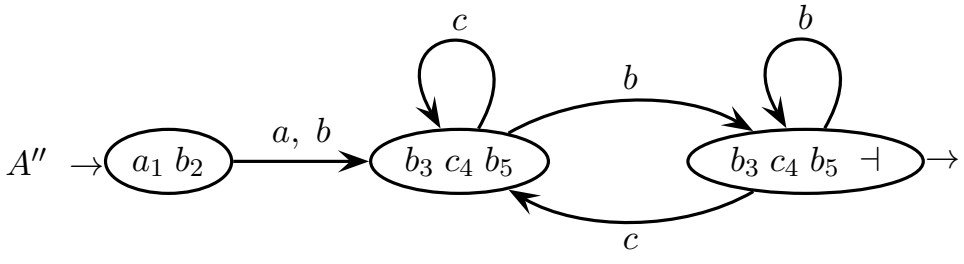


Automaton A' is still nondeterministic, on state (2 3 4), but no longer ambiguous. For completeness, here is an equivalent regular expression $R = (\,a \mid b\,)(\,b \mid c\,)^*\,b = (\,a \mid b\,)(\,c^*\,b\,)^+$, easily obtained by eliminating the middle node of A'.

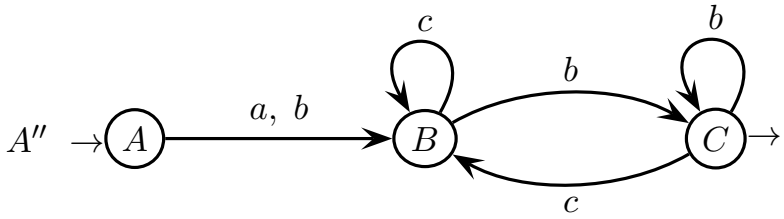
(c) Here is the result of the Berry-Sethi algorithm (BS), applied to automaton A':



initials	$a_1\ b_2$
generators	followers
a_1	$b_3\ c_4\ b_5$
b_2	$b_3\ c_4\ b_5$
b_3	$b_3\ c_4\ b_5$
c_4	$b_3\ c_4\ b_5$
b_5	$\dashv$



With a change of state names, the deterministic automaton A'' becomes so:



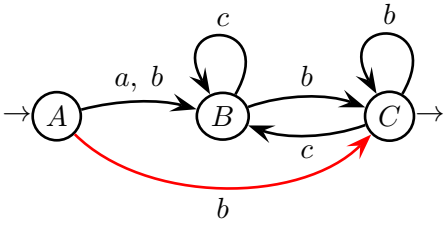
The deterministic automaton A'' is minimal: the final state C is distinguishable from the two non-final states A and B, and state B is distinguishable from state A because transition $\delta(B, c)$ is defined, whereas transition $\delta(A, c)$ is not so.

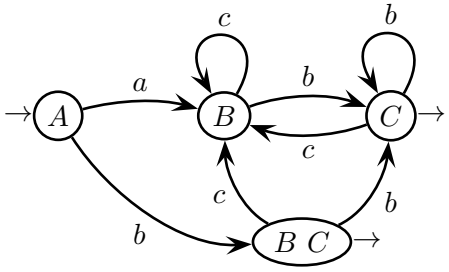
(d) Test of automaton A'': $A \xrightarrow{a} B \xrightarrow{b} C$, $A \xrightarrow{b} B \xrightarrow{b} C$, etc (the reader may complete the test autonomously); all the short strings of question (a) are accepted.

(e) Language $L(A)$ is not local. In fact, string $b\,b \in L(A)$, thus for local sets of $L(A)$ it holds $b \in \mathit{Ini}$ and $b \in \mathit{Fin}$. Yet string $b \notin L(A)$, though it complies with the local sets as well, as it starts and ends by b, and it contains no digram (thus no forbidden digram either). This concludes the question.

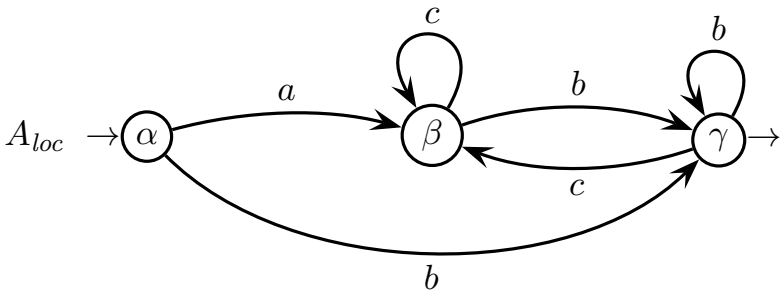
A proof can be also obtained through the property that the minimal automaton of a local language has to be local. Automaton A'' is minimal, but not local, as label b enters two different states. Thus language $L(A'') = L(A)$ is not local.

For more insight, in some sense language $L(A)$ is close to being local: just add string b to it, and it becomes local. Here is why: first add the arc $A \xrightarrow{b} C$ (red) to automaton A'', which is equivalent to automaton A, then determinize:





The two final states states (B C) and C are undistinguishable. Thus merge them and rename the states. Consider the resulting automaton A_{loc} below:



Automaton A_{loc} is clearly local (each arc label enters only one state, namely β for a and c, and γ for b) and normalized (the initial state α does not have any ingoing arcs). By construction, it holds $L(A_{loc}) = L(A'') \cup \{ b \} = L(A) \cup \{ b \}$.

2 Free Grammars and Pushdown Automata 20%

1. The following grammar G, over a two-letter terminal alphabet $\{ a, b \}$, generates all and only the strings that contain the same number of letters a and b (axiom S):

$$G \begin{cases} S \rightarrow a\, S\, b \\ S \rightarrow b\, S\, a \\ S \rightarrow S\, S \\ S \rightarrow \varepsilon \end{cases}$$

Answer the following questions:

(a) Verify that grammar G is ambiguous, by drawing at least two syntax trees for the valid string:

$$a\ b\ a\ b$$

Find the ambiguity degree of the above string and adequately justify your answer.

(b) Modify grammar G and obtain a new grammar G' (possibly ambiguous), such that (in addition to the original property that the number of letters a and the number of letters b are the same over the entire string) for every prefix x of each string $s \in L(G')$, this inequality holds:

$$\big|\, |\, x\, |_a - |\, x\, |_b \,\big| \leq 1$$

In words, the modulus (absolute value) of the difference between the numbers of letters a and b in the prefix x is smaller than or equal to 1.

Sample valid strings:

$$a\ b\ b\ a \qquad b\ a\ a\ b \qquad a\ b\ a\ b \qquad b\ a\ b\ a$$

Sample invalid strings:

$$b\ b\ a\ a \qquad a\ a\ b\ b$$

In modifying grammar G, you may add new nonterminals, but keep in mind that simple solutions are preferred.

(c) Modify grammar G and obtain a new grammar G'' (possibly ambiguous), such that (in addition to the original property that the number of letters a and the number of letters b are the same over the entire string) for every prefix x of each string $s \in L(G'')$, this inequality holds:

$$|\, x\, |_b \geq |\, x\, |_a$$

Sample valid strings:

$$b\ a\ b\ a \qquad b\ b\ a\ a \qquad b\ b\ a\ b\ a\ a$$

Sample invalid strings:

$$b\ a\ a\ b \qquad a\ b\ b\ a \qquad b\ a\ b\ a\ a\ b$$

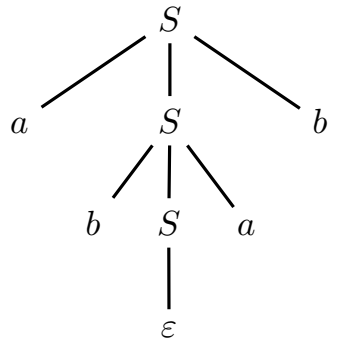
Keep in mind that simple solutions are preferred.

(d) (optional) If grammar G'' is ambiguous, write an equivalent non-ambiguous grammar G'''.

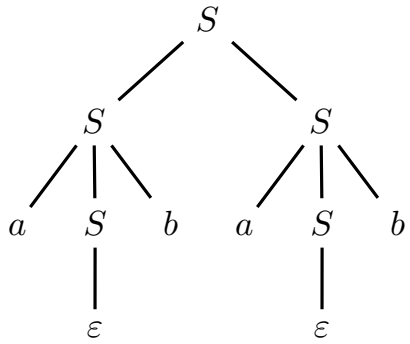
Solution

(a) Here are two derivations (both leftmost) for the valid string $a\,b\,a\,b$ of grammar G (the nonterminal to derive is underlined), and their syntax trees below:

$\underline{S} \Rightarrow a\ \underline{S}\ b \Rightarrow a\ b\ \underline{S}\ a\ b \Rightarrow a\ b\ a\ b$



$\underline{S} \Rightarrow \underline{S}\ S \Rightarrow a\ \underline{S}\ b\ S \Rightarrow a\ b\ \underline{S}$
$\Rightarrow a\ b\ a\ \underline{S}\ b \Rightarrow a\ b\ a\ b$



The ambiguity degree of string $a\,b\,a\,b$ is infinite, due to the alternative rules $S \to S\,S \mid \varepsilon$ (two-sided recursion) that allow the empty string ε to be derived in an infinite number of ways. In particular, grammar G has a two-step circular derivation $S \overset{S \to S\,S}{\Longrightarrow} S\,S \overset{S \to \varepsilon}{\Longrightarrow} S$, which allows any tree branch that contains an inner node S to be arbitrarily stretched, with no effect on the tree frontier.

(b) Here is a grammar G' such that for every prefix x of each string $s \in L(G')$, i.e., for every x with $s = x\,y$ for some $y \in \Sigma^+$, the right inequality holds (axiom S):

$$G' \begin{cases} S \to a\ S_b\ b \mid b\ S_a\ a \mid S\ S \mid \varepsilon \\ S_b \to b\ S\ a \mid \varepsilon \\ S_a \to a\ S\ b \mid \varepsilon \end{cases} \qquad \big|\,|\,x\,|_a - |\,x\,|_b\,\big| \leq 1$$

Grammar G' is a restriction of grammar G, where certain derivations of G are impossible. Thus, globally the numbers of letters a or b are the same, as all the derivations of G already have this property. The restriction is that whenever nonterminal S extends the current prefix with a letter a, nonterminal S_b soon appends a letter b (or ε) next to a, thus preserving the inequality; similarly for letter b. Also grammar G' is ambiguous, for the same reasons as for G.

Grammar G' derives from grammar G and retains most structure of G, which is based on nesting and concatenating long-distance balanced letter pairs. Here is a second approach: by considering the language, a valid string must have an even length and it can be modeled as a list of elements of type $a\,b$ and $b\,a$, i.e., short-distance balanced letters pairs. In fact, only so in every prefix the excess of a letter type over the other type is limited to 1 (the valid sample strings also hint so) and is 0 at the string end. Thus a grammar variant G'_v is (axiom S):

$$G'_v\colon S \to a\ b\ S \mid b\ a\ S \mid \varepsilon$$

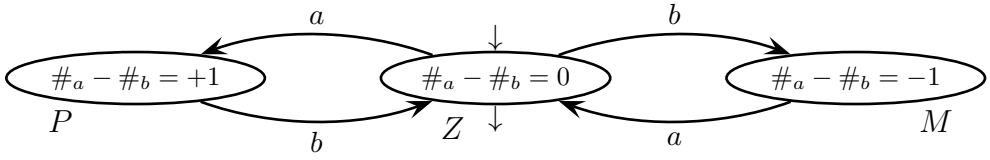
equivalent to grammar G', i.e., $L(G'_v) = L(G')$, and non-ambiguous. Grammar G'_v is right-linear, thus the language is regular, a finding that is not obvious from observing G'. Of course, grammar G'_v is structurally very different from G'.

Furthermore, grammar G'_v can be compacted in the following form (axiom S):

$$G'_v\colon S \to S\ S \ \mid\ a\ b \ \mid\ b\ a \ \mid\ \varepsilon$$

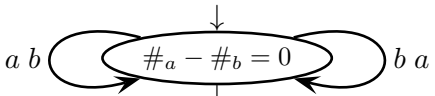
It can still produce a list of elements of type $a\ b$ and $b\ a$, though ambiguously.

A third approach considers an automaton that recognizes the language, and then transforms such an automaton into a grammar. The only language requirements are the inequality $\big|\,|\,x\,|_a - |\,x\,|_b\,\big| \leq 1$, to be fulfilled whenever the automaton reads an input letter, and the equality $|\,x\,|_a = |\,x\,|_b$ at the input end. Thus, if we respectively term $\#_a$ and $\#_b$ the numbers of letters a and b currently read in the input, the automaton should only keep memory of their difference, say $\#_a - \#_b$. From the requirements, the difference can only take the three values -1, 0 and $+1$, and it must take the value 0 at the input end. This can be accomplished with a finite memory, thus the automaton is finite-state (three states) and of course the language is regular (again a non-obvious finding). Here is the automaton:



Initially the difference is 0. The (deterministic) automaton can be transformed into a (non-ambiguous) right-linear grammar variant G'_{vv}, equivalent to grammar G', i.e., $L\,(G''_{vv}) = L\,(G')$. Name the states P (plus), Z (zero) and M (minus). Here is grammar G'_{vv} (axiom Z – left):

$$G'_{vv}\ \begin{cases} Z \ \to\ a\ P \ \mid\ b\ M \ \mid\ \varepsilon \\ P \ \to\ b\ Z \\ M \ \to\ a\ Z \end{cases}$$



$$Z \to a\ b\ Z \ \mid\ b\ a\ Z \ \mid\ \varepsilon$$

By eliminating nodes P and M, the generalized automaton above (right top) is obtained, and its grammar (right bottom) is G'_v, after renaming Z into S.

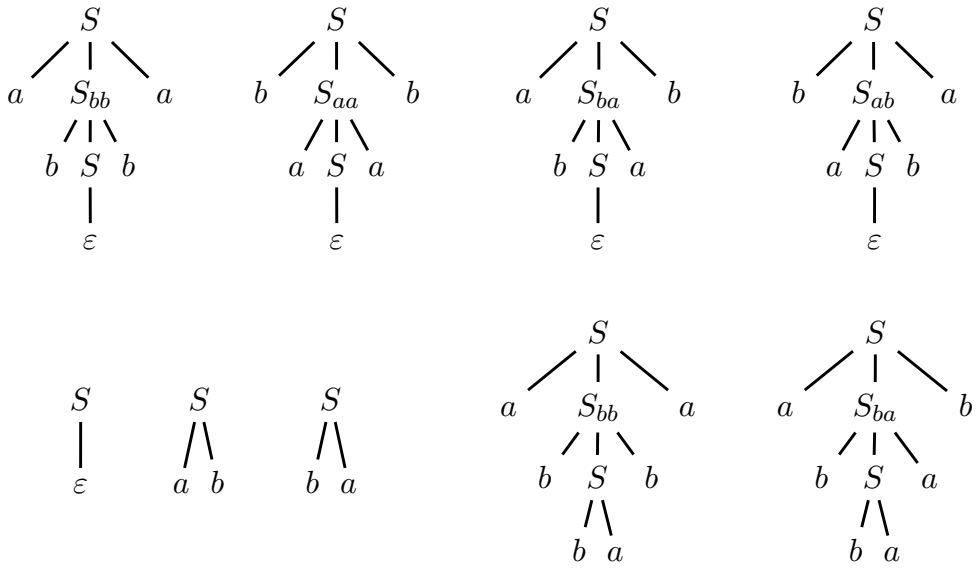
These are three possible approaches, which yield structurally different solutions, depending on how far we go from the original structure presented through grammar G. Grammar G' is the one structurally nearest to G, and grammar G'_{vv} the farthest. Of course, there may exist even more solutions, with new structures.

We just give one more solution, which is a new variant of grammar G'. Since a valid string of language $L\,(G')$ must have an even length, we can model the string as one nest of the pairs $a\,a$, $a\,b$, $b\,a$ and $b\,b$, which are all those possible with two letters, provided that we do not allow the string to be globally unbalanced, or to contain more than two consecutive identical letters, or to start or end with two identical letters. Instead, we never need to concatenate any two such pairs.

Thus, a new grammar variant G'_{vvv}, equivalent to G', is the following (axiom S):

$$G'_{vvv} \begin{cases} S \to a\ S_{bb}\ a \mid a\ S_{ba}\ b \mid b\ S_{ab}\ a \mid b\ S_{aa}\ b \mid a\ b \mid b\ a \mid \varepsilon \\ S_{bb} \to b\ S\ b \\ S_{ba} \to b\ S\ a \\ S_{ab} \to a\ S\ b \\ S_{aa} \to a\ S\ a \end{cases}$$

The reader can verify it autonomously. The valid strings and trees below help:



Notice that grammar G'_{vvv} does not make use of a rule for concatenating letter pairs, like $S \to S\,S$, but that it can only nest them. Comparing with grammar G', we see that the idea of using letter pairs is retained, yet by only nesting them, not by concatenating them, and that consequently the pairs used must include (beside $a\,b$ and $b\,a$) also the new ones $a\,a$ and $b\,b$, which are absent from G'. Thus grammar G'_{vvv} is somehow transversal to grammar G': it retains part of the structure of G' and adds some new structure, which is not part of G'.

Furthermore, grammar G'_{vvv} is non-ambiguous, as the derivation step where to use a rule is uniquely determined by the corresponding letter pair. This new version G'_{vvv} is somewhat intricate, compared to the direct previous ones.

(c) Here is a grammar G'' such that for every prefix x of each string $s \in L\,(G'')$, the inequality on the right holds (axiom S):

$$G'' \begin{cases} S \to b\ S\ a \\ S \to S\ S \\ S \to \varepsilon \end{cases} \qquad |\,x\,|_b \geq |\,x\,|_a$$

Grammar G'' is the same as grammar G, but rule $S \to a\ S\ b$ is removed, thus clearly ensuring that the inequality is fulfilled for every string prefix. Notice that also grammar G'' is ambiguous, again for the same reasons as for G.

(d) We cannot conclude that language $L(G'')$ is regular, as we were able to do for language $L(G')$. In fact, now the difference of the numbers of letters a or b in a string prefix is unbounded, which is not in the scope of a finite-memory device. Anyway, this reasoning alone is insufficient to disprove the regularity of the language (maybe the difference is inessential), though it leaves little hope.

Therefore, it is advisable to consider whether language $L(G'')$ may be some already known context-free non-regular language. Actually, after quickly examining the structure of grammar G'', it easily turns out that language $L(G'')$ is Dyck with just one parenthesis type, where the symbols b and a respectively represent a left (open) and a right (closed) parenthesis (the valid sample strings also hint so). This finding implies that certainly the language is not regular, as we have already suspected intuitively.

Therefore, quite simply, a well-known non-ambiguous grammar variant for $L(G'')$ is the following (axiom S):

$$G''_v\colon S \to b\ S\ a\ S \ \mid\ \varepsilon$$

As for question (b), a simpler (and well-known) grammar variant is obtained by directly reasoning on the language structure, instead of restricting the original grammar G. Consequently, the grammar structure changes significantly.

2. Consider a script language that defines two-dimensioned tables, specified as follows:

   - a script defines a series of one or more table definitions, without separator
   - a table definition has a title and contents
   - the table contents consist of a series of one or more columns, separated by semicolon, and each column has a title and contents
   - the column contents are a list of one or more values (as defined below), separated by comma
   - a group of consecutive columns (one or more) may be covered (its contents are not visualized) by using the keywords hide-unhide; two or more groups of columns, adjacent or separated, may be covered
   - a group of consecutive values in a column may be covered (same as before) by using the keywords mask-unmask
   - a title (table or column) is an identifier; a value can be a number or a subtable
   - a subtable is like a table, but it does not have a title (only its columns are titled)

   Consider the following sample table, with some covered parts, and its script below:

<table>
<tr><th colspan="4">ExampleTableTitle</th></tr>
<tr><th colspan="2">alpha</th><th>beta</th><th>gamma</th><th>delta</th></tr>
<tr><td colspan="2">5</td><td>–</td><td>–</td><td>7</td></tr>
<tr><td colspan="2"><table><tr><th>epsilon</th><th>rho</th></tr><tr><td>11</td><td>13</td></tr><tr><td>–</td><td>17</td></tr></table></td><td>–</td><td>–</td><td>–</td></tr>
<tr><td colspan="2">9</td><td>–</td><td>–</td><td>–</td></tr>
</table>

```
table ExampleTableTitle contents are
  column alpha is
    5,
    subtable contents are
      column epsilon is 11, mask 15 unmask endcol;
      column rho     is 13, 17 endcol
    endsubtbl,
    9
  endcol;
  hide
    column beta  is 30, 31, 32 endcol;
    column gamma is 40, 41, 42 endcol
  unhide;
  column delta is 7, mask 8, 9 unmask endcol
endtbl
```

You should infer from this sample script any information that is not explicitly given above, in particular the symbols for keywords, delimiters and their placement, etc.

Write a grammar, possibly of type *EBNF* and not ambiguous, that generates the script language described above. *In the grammar, schematize the identifiers and numbers by terminals* id *and* num*, respectively, without expanding them with rules.*

Solution

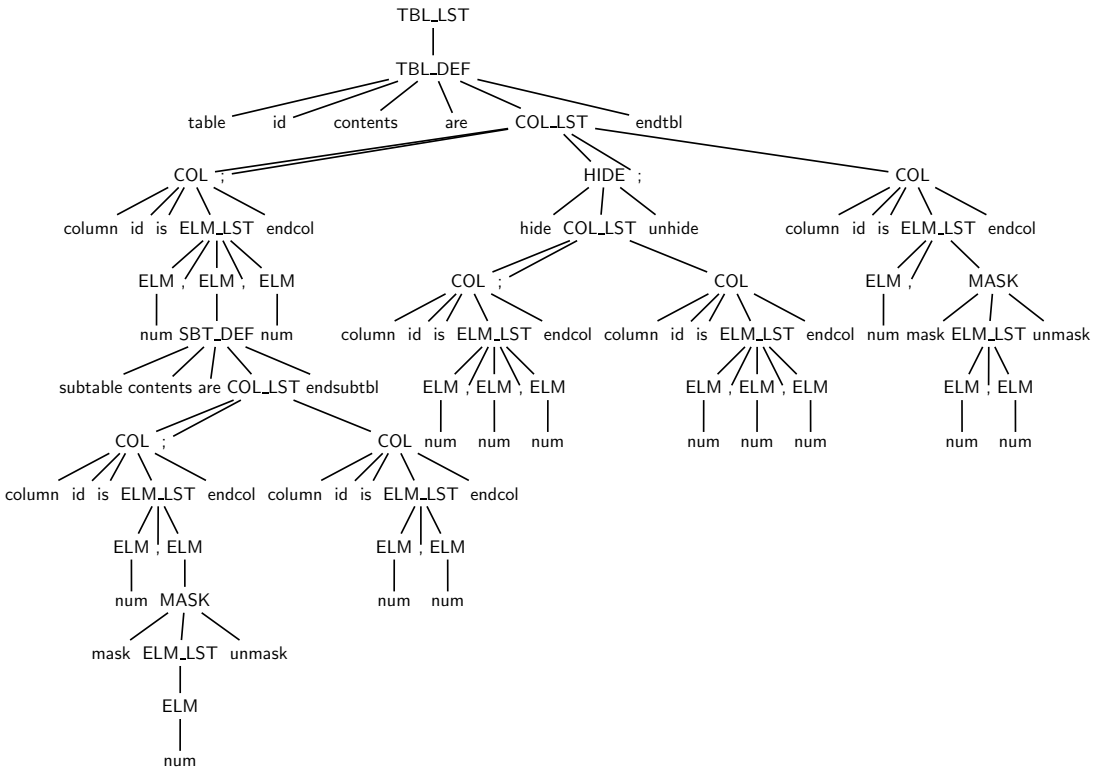
Here is a viable grammar, of type *EBNF* and not ambiguous (axiom TBL_LST):

$$
\begin{cases}
\langle \text{TBL\_LST} \rangle \;\to\; \langle \text{TBL\_DEF} \rangle^{+} \\
\hline
\langle \text{TBL\_DEF} \rangle \;\to\; \text{table id contents are } \langle \text{COL\_LST} \rangle \text{ endtbl} \\
\hline
\langle \text{COL\_LST} \rangle \;\to\; \big(\langle \text{COL} \rangle \mid \langle \text{HIDE} \rangle \big) \; \big(\text{`;'} \; \big(\langle \text{COL} \rangle \mid \langle \text{HIDE} \rangle \big) \big)^{*} \\
\hline
\langle \text{COL} \rangle \;\to\; \text{column id is } \langle \text{ELM\_LST} \rangle \text{ endcol} \\
\langle \text{HIDE} \rangle \;\to\; \text{hide } \langle \text{COL\_LST} \rangle \text{ unhide} \\
\hline
\langle \text{ELM\_LST} \rangle \;\to\; \langle \text{ELM} \rangle \; \big(\text{`,'} \; \langle \text{ELM} \rangle \big)^{*} \\
\hline
\langle \text{ELM} \rangle \;\to\; \text{num} \mid \langle \text{MASK} \rangle \mid \langle \text{STB\_DEF} \rangle \\
\hline
\langle \text{MASK} \rangle \;\to\; \text{mask } \langle \text{ELM\_LST} \rangle \text{ unmask} \\
\langle \text{STB\_DEF} \rangle \;\to\; \text{subtable contents are } \langle \text{COL\_LST} \rangle \text{ endsubtbl}
\end{cases}
$$

This grammar can freely nest hide and mask blocks (such a feature was left optional). Likewise, most editors can cover graphic objects with multiple layers, and uncover them one layer at a time. Thus the grammar enables such a common editing function. However, we could slightly change the grammar (possibly by adding a few nonterminals) and disallow such a nesting. We could even categorize more, with these rules:

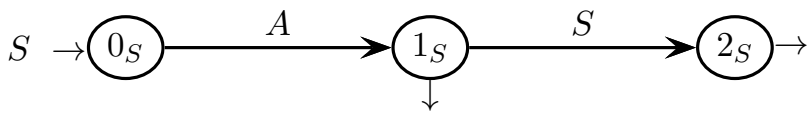
$$
\begin{aligned}
\langle \text{COL\_LST} \rangle &\;\to\; \langle \text{GROUP} \rangle \; \big(\text{`;'} \; \langle \text{GROUP} \rangle \big)^{*} \\
\langle \text{GROUP} \rangle &\;\to\; \langle \text{COL} \rangle \mid \langle \text{HIDE} \rangle
\end{aligned}
$$

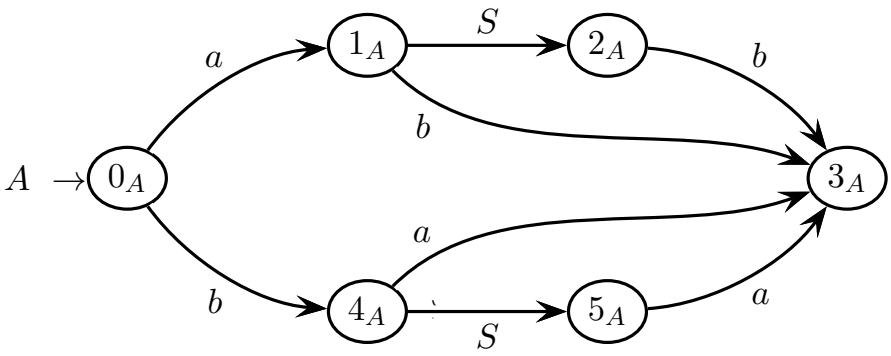
For completeness, here is the syntax tree of the sample script above:



3 Syntax Analysis and Parsing Methodologies 20%

1. Consider the grammar G below, represented as a machine net, over a two-letter terminal alphabet $\Sigma = \{\ a,\ b\ \}$ and a two-symbol nonterminal alphabet $V = \{\ S,\ A\ \}$ (axiom S):





Answer the following questions (use the figures / tables / spaces on the next pages):

(a) Draw a minimal portion of the pilot of grammar G, sufficient to show that grammar G is not $ELR(1)$, and suitably explain why the $ELR(1)$ property fails.

(b) Write all the guide sets of the machine net of grammar G (on the call arcs and exit arrows), and discuss whether grammar G is $ELL(1)$. In the case it is not so, list all the guide set conflicts that are present.

(c) By using the Earley method, analyze the following valid string:

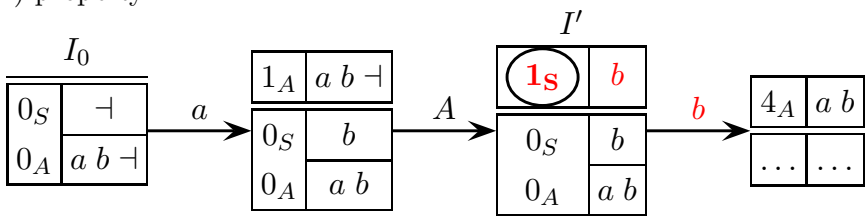
$$a\ b\ a\ b \in L\ (G)$$

On the Earley vector, highlight the item(s) that determine the acceptance of the above string and possibly of any string prefix.

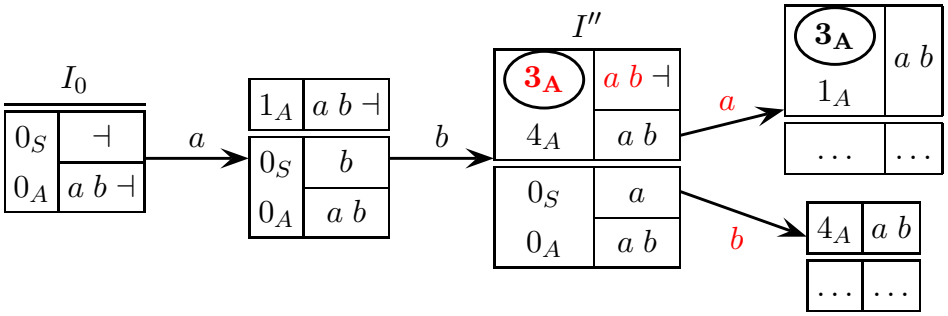
(d) (optional) Draw any possible syntax tree of the string analyzed at point (c).

Solution

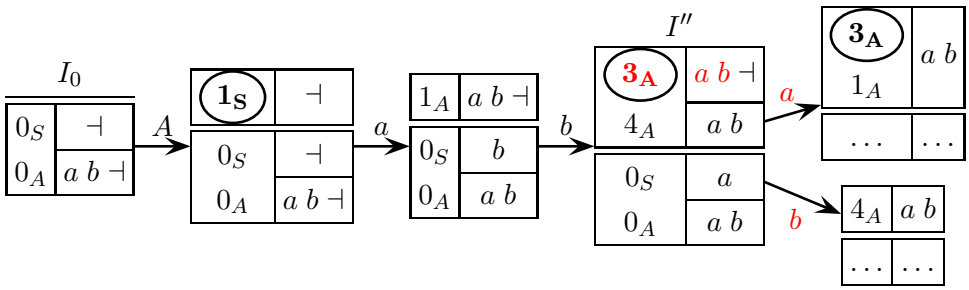
(a) Here is a small portion of the pilot of grammar G, sufficient to disprove the $ELR(1)$ property:



There is a shift-reduce conflict that involves the final item $\langle\, 1_S,\, b \,\rangle$ in I', on terminal b (red). Another portion, with different conflicts, is the following:



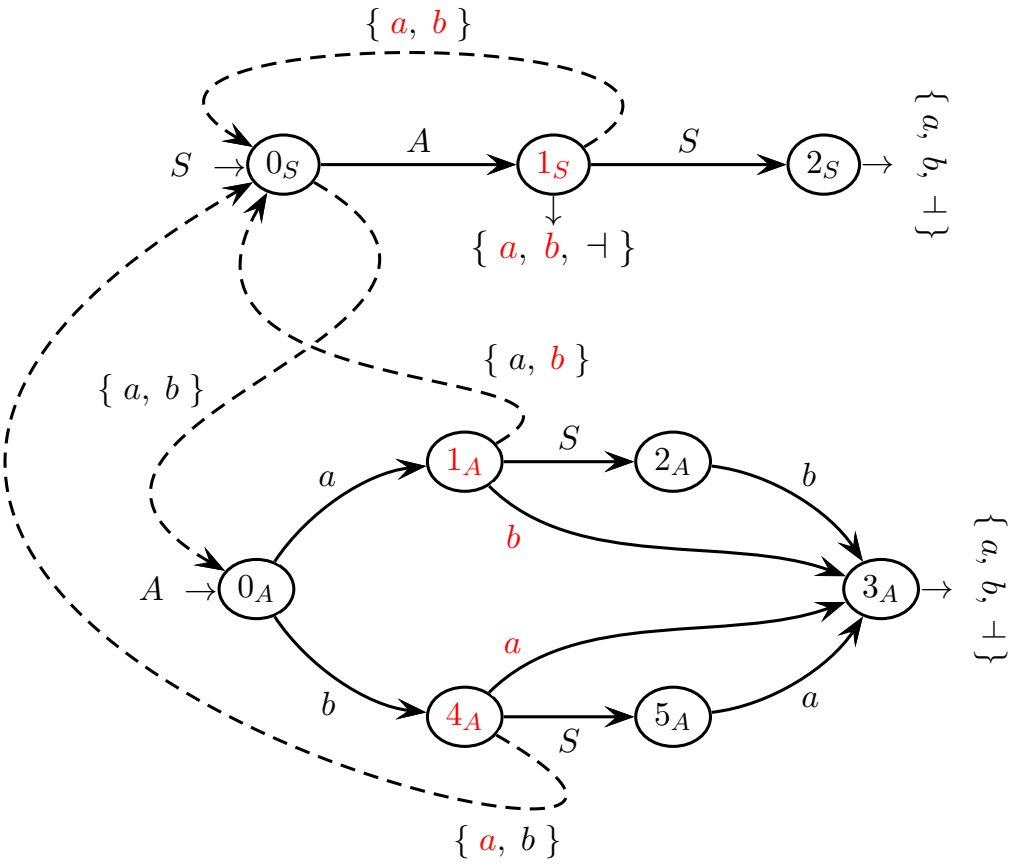
There are two shift-reduce conflicts that involve the final item $\langle\, 3_A,\, a\, b\, \dashv \,\rangle$ in I'', on terminals a and b (red). One more portion, with a partially different path from I_0 but the same two conflicts in I'', is the following:



Therefore, clearly grammar G is not $ELR(1)$. There may be more conflicts in the entire pilot, possibly of type different from shift-reduce. For instance, since machine M_A has a symmetry axis (imagine to flip it vertically), which exchanges terminals a-b and states 1-4 2-5, while states 1 and 3 (and the whole machine M_S) are left unchanged, all the conflicts (of whatever type) that involve such elements are duplicated. As an example related to the first portion, the transition sequence $I_0 \xrightarrow{b} \dots \xrightarrow{A} \dots \xrightarrow{a} \dots$ would lead to the symmetrical conflicts. Actually, it turns out that grammar G is ambiguous (see questions (c) and (d)), thus the pilot is expected to have conflicts (though analysis is required for their type).

The pilot does not have the STP, since for instance the first b-transition in the second portion is multiple (double – but not convergent). Also the third portion has a multiple (not convergent) b-transition. There are many others, too.

(b) Here are all the guide sets on the call arcs and exit arrows of the machine net of grammar G:

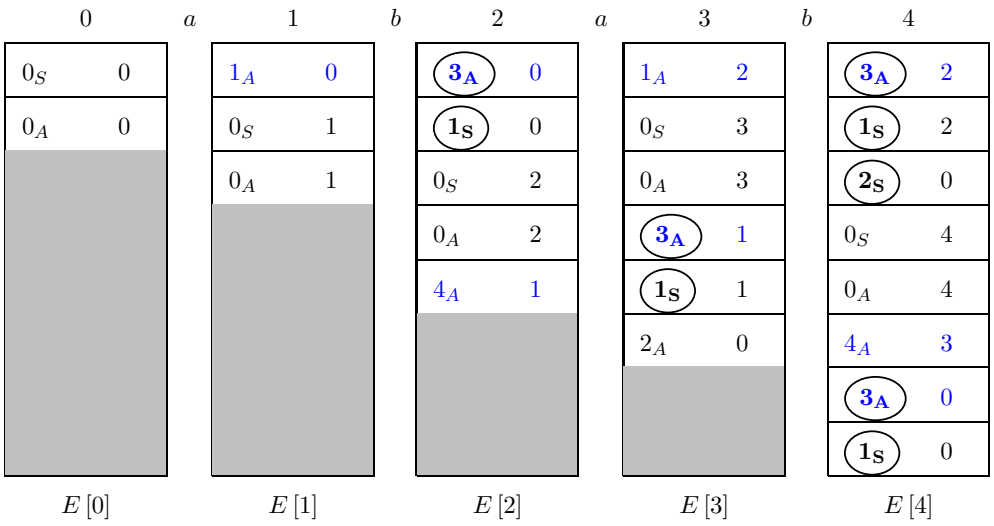


There are guide set conflicts on the (red) bifurcation states 1_S (on terminals a and b), 1_A (on terminal b) and 4_A (on terminal a). Thus grammar G is not $ELL(1)$. This is coherent with the fact that the pilot does not have the *STP*, see question (a), and that grammar G is ambiguous, see questions (c) and (d).

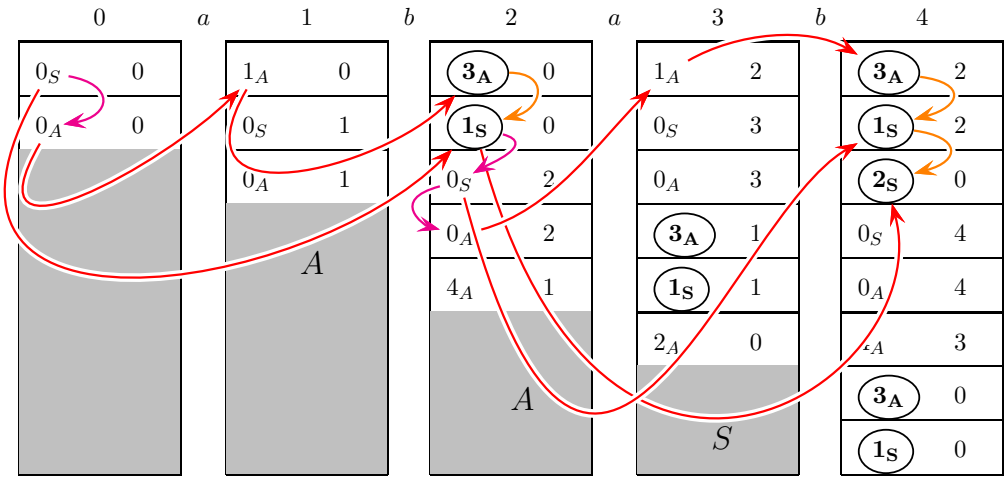
For generality, we verify that the *ELL* property fails with a look-ahead window of *any* size $k \geq 2$, as we expect to happen since grammar G is ambiguous. In fact, nonterminal S can generate any number of consecutive letters a, e.g., $S \Rightarrow A \Rightarrow a\ S\ b \Rightarrow a\ A\ b \Rightarrow a^2\ S\ b^2 \Rightarrow \ldots \Rightarrow a^k\ S\ b^k$, and it can be followed by any number of letters a, e.g., $S \Rightarrow A \Rightarrow b\ S\ a \Rightarrow b\ A\ a \Rightarrow b^2\ S\ a^2 \Rightarrow \ldots \Rightarrow b^k\ S\ a^k$. Thus, on state 1_S, beside the already known conflict on terminal a (and b), there is a conflict on string a^k for any $k \geq 2$ (and the same happens with b^k).

Therefore grammar G is definitely not $ELL(k)$, for any $k \geq 1$. Of course, language $L(G)$ might admit another grammar G', non-ambiguous or even deterministic *ELL* or *ELR*. The reader may try to design it autonomously.

(c) Here is the complete Earley vector for the valid sample string $a\,b\,a\,b \in L\,(G)$:

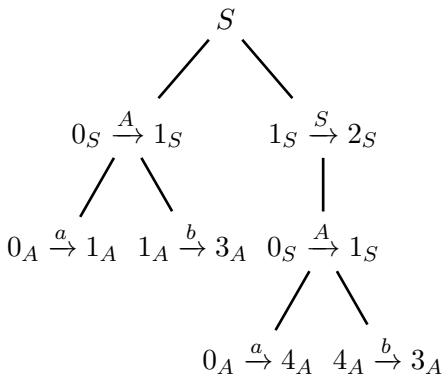


Remember that a vector state $E\,[i]$ is an unordered list of items ($0 \leq i \leq 4$). We unavoidably order the items when we list them, yet we can do so arbitrarily. Thus, to help the reader understand the vector, we colour blue the items in state $E\,[i]$ that are obtained from state $E\,[i-1]$ through terminal shift ($1 \leq i \leq 4$). The string is accepted and has two different derivations, due to the two final axiomatic items with pointer 0 in the state $E\,[4]$, i.e., $\langle\, 2_S,\ 0\,\rangle$ and $\langle\, 1_S,\ 0\,\rangle$; see question (d) for their syntax trees. Thus the string and grammar G itself are ambiguous. The prefix $a\,b$ is accepted as well, due to the final axiomatic item with pointer 0 in the state $E\,[2]$, i.e., $\langle\, 1_S,\ 0\,\rangle$; see also question (d) for its tree. Here is the reconstruction of the tree of string $a\,b\,a\,b$, from item $\langle\, 2_S,\ 0\,\rangle$:

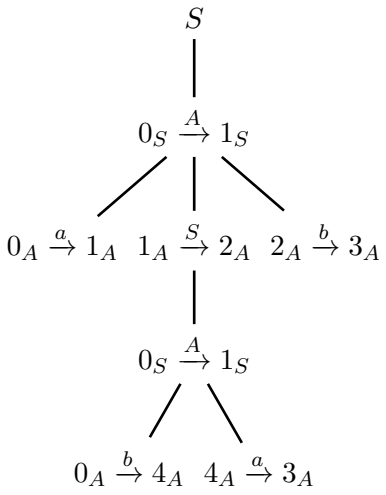


Magenta (pink), red and orange colours respectively represent closure, shift (terminal and non) and completion (reduction) steps, in Earley terminology. The three nonterminal shift arcs are labeled with the nonterminals shifted on, respectively, corresponding to the three inner nodes of the syntax tree. Reconstructing the other tree, as well as the tree of the accepted prefix, is left to the reader.

(d) Here are the syntax trees of the valid string $a\,b\,a\,b \in L\,(G)$, with machine paths:

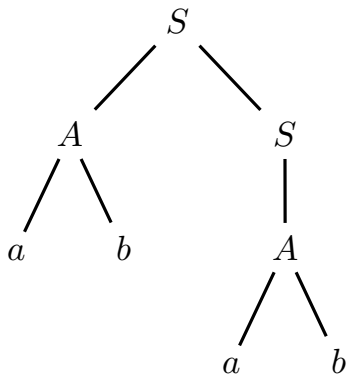


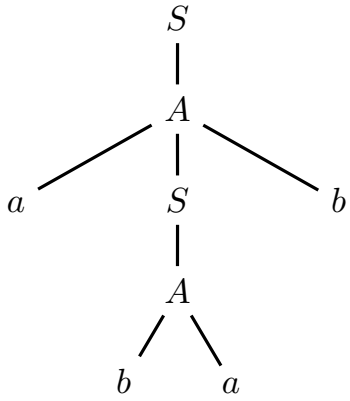
from item $\langle\, 2_S,\ 0\,\rangle$ in $E\,[4]$



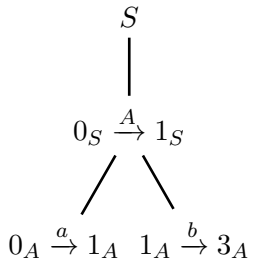
from item $\langle\, 1_S,\ 0\,\rangle$ in $E\,[4]$

Here are their projections over the grammar (terminal and non) alphabets:

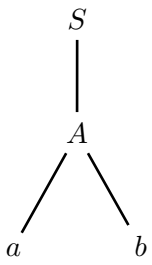




These two are the only syntax trees of string $a\,b\,a\,b$. Finally, here is also the single syntax tree of the accepted prefix $a\,b$, which is not ambiguous:



from item $\langle\, 1_S,\ 0\,\rangle$ in $E\,[2]$



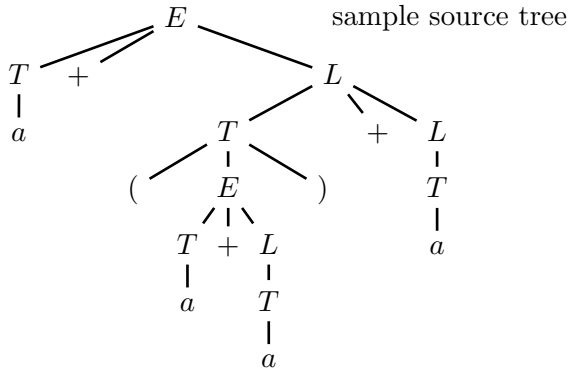
projection over the grammar alphabets

There are not any other trees for string $a\,b\,a\,b$, or recognized string prefixes.

4 Language Translation and Semantic Analysis 20%

1. Consider the source grammar G_s below, for the arithmetic expressions with infix addition $+$, round brackets (subexpressions), and variables schematized by terminal a (axiom E):

$$G_s \begin{cases} E \to T + L \\ L \to T + L \\ L \to T \\ T \to (E) \\ T \to a \end{cases}$$



Sample source strings are: $a+a$, $a+a+a$, $a+(a+a)+a$ (syntax tree above right).

Consider a translation that converts addition into prefix form with multiple addends (possibly more than two). The idea is to convert $a+a+a$ into $+3\ a\ a\ a$, for instance, where number 3 specifies the actual number of addends.

Here we want to design a syntactic translation function τ that produces such a multiple addend prefix form, though it does not compute the number of addends. Instead, it outputs a placeholder schematized by terminal n, e.g., $\tau\,(a+a+a) = +n\ a\ a\ a$. A semantic translation, *not to be considered here*, will compute the actual number of addends and will substitute such a number to the placeholder n.

See these examples (the actual number of addends is shown below n as a comment):

$$\begin{aligned} \tau\,(a+a) &= +\underset{2}{n}\ a\ a \\ \tau\,(a+a+a) &= +\underset{3}{n}\ a\ a\ a \\ \tau\,(a+(a+a)+a) &= +\underset{3}{n}\ a\ +\ \underset{2}{n}\ a\ a\ a \qquad \text{– translation } \tau \text{ of the example} \end{aligned}$$

Answer the following questions:

(a) Write a translation scheme G_τ that defines the syntactic translation function τ described above. Do not change the source grammar G_s. Test scheme G_τ by drawing the syntax translation tree of the sample translation τ above.

(b) Write a translation scheme $G_{\tau'}$ for an optimized syntactic translation τ' that does not output the placeholder n if addition has only two operands, as follows:

$$\tau'\,(a+(a+a)+a) \;=\; +\underset{3}{n}\ a\ +\ a\ a\ a \qquad \text{– sample translation } \tau'$$

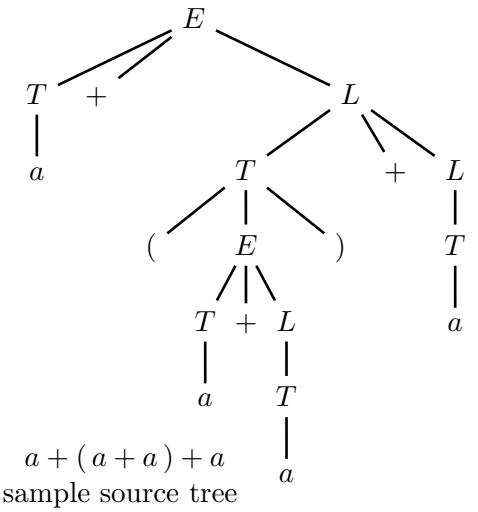
For τ' you may change the source grammar G_s, if necessary. Test scheme $G_{\tau'}$ by drawing the syntax translation tree of the sample translation τ' above.

(c) (optional) Determine whether translations τ and τ' can be computed deterministically by an $ELL(k)$ parser with write actions, for suitable values of $k \geq 1$, and motivate your answers (yes or no).

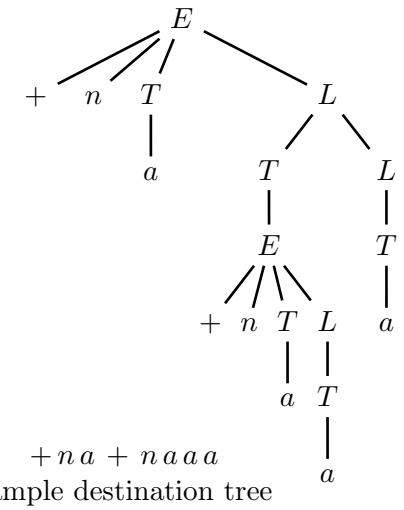
Solution

(a) Here is scheme G_τ (axiom E), with both the source and destination grammars G_s and G_d, and their sample source and destination trees below:

$$G_s \begin{cases} E \to T + L \\ L \to T + L \\ L \to T \\ T \to (E) \\ T \to a \end{cases} \qquad G_d \begin{cases} E \to + n\, T\, L \\ L \to T\, L \\ L \to T \\ T \to E \\ T \to a \end{cases}$$



$a + (a + a) + a$
sample source tree



$+ n\, a\, +\, n\, a\, a\, a$
sample destination tree

Scheme G_τ is essentially the classical syntactic translation of arithmetic expressions from infix form to prefix form, though here applied to n-ary addition ($n \geq 2$). The compound sign "$+n$" is prepended to the entire operand list, and the infix sign(s) "$+$" is (are) canceled. This is recursively carried on for each subexpression, whereas all the surrounding round brackets are canceled.

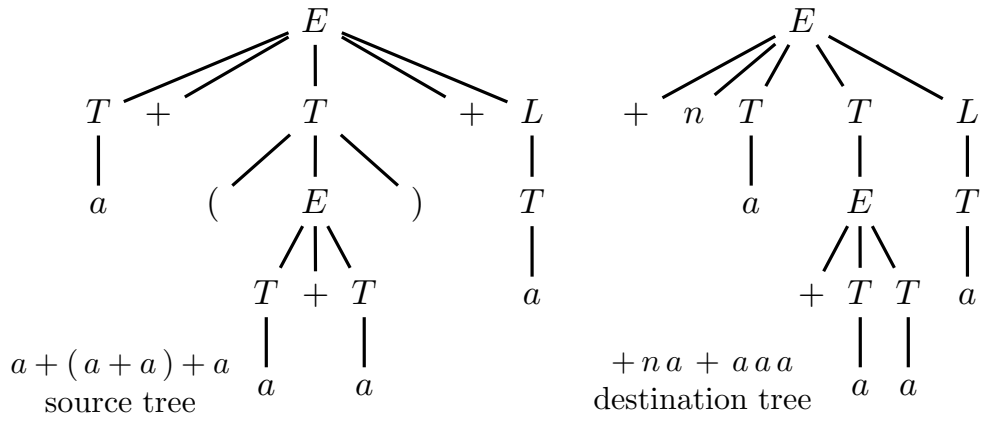
For completeness, here is the combined scheme G_τ (axiom E), on the left:

$$G_\tau \begin{cases} E \to \frac{\varepsilon}{+n}\, T\, \frac{+}{\varepsilon}\, L \\ L \to T\, \frac{+}{\varepsilon}\, L \\ L \to T \\ T \to \frac{(}{\varepsilon}\, E\, \frac{)}{\varepsilon} \\ T \to \frac{a}{a} \end{cases} \qquad G_\tau^{EBNF} \begin{cases} E \to \frac{\varepsilon}{+n}\, T\, \left(\frac{+}{\varepsilon}\, T\right)^+ \\ T \to \frac{(}{\varepsilon}\, E\, \frac{)}{\varepsilon} \\ T \to \frac{a}{a} \end{cases}$$

The source and destination trees can be overlapped accordingly. For curiosity, on the right it is shown an *EBNF* scheme G_τ^{EBNF} (axiom E), equivalent to the *BNF* scheme G_τ. The placeholder n should be assigned (by a semantic translation) the number of iterations of the cross operator, incremented by 1.

(b) Here is scheme $G_{\tau'}$ (axiom E), with both the source and destination grammars $G_{s'}$ and $G_{d'}$, and their source and destination trees below ($n \geq 3$):

$$G_{s'} \begin{cases} \text{split rule} \begin{cases} E \to T + T & \text{– binary} \\ E \to T + T + L & \text{– } n\text{-ary} \end{cases} \\ \text{same as in } G_s \begin{cases} L \to T + L \\ L \to T \\ T \to (E) \\ T \to a \end{cases} \end{cases} \qquad G_{d'} \begin{cases} E \to + T\, T \\ E \to + n\, T\, T\, L \\ L \to T\, L \\ L \to T \\ T \to E \\ T \to a \end{cases}$$



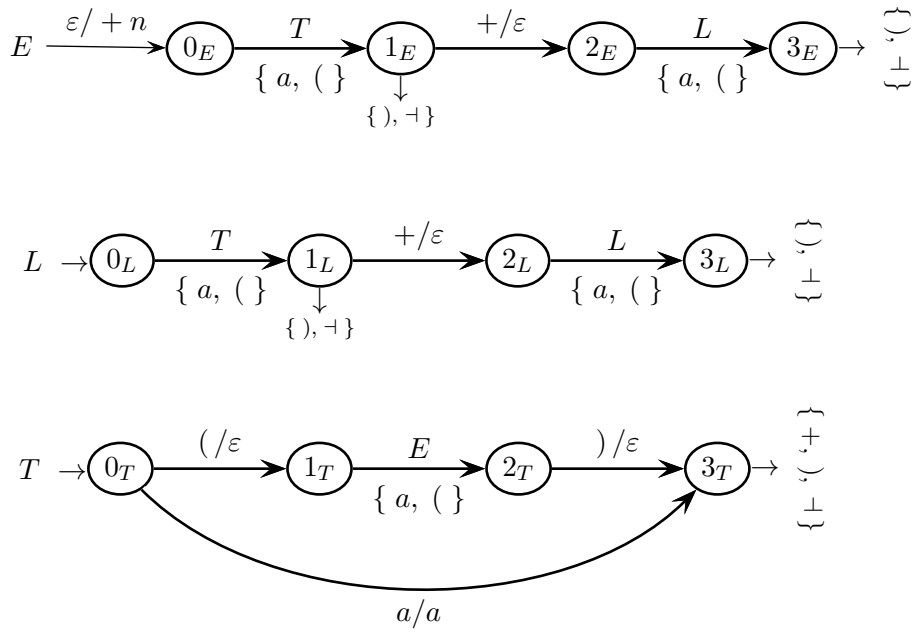
Scheme $G_{\tau'}$ is a variant of the previous scheme G_τ, where the source grammar is slightly changed. The idea is to distinguish between binary addition and n-ary addition with $n \geq 3$, by means of two alternative rules for nonterminal E. The former rule is for binary addition, the latter for ternary or more. In the destination grammar both rules prepend a sign "+" to the operand list, but only the rule that generates n-ary addition in the source grammar additionally prepends the placeholder n. The rest of scheme $G_{\tau'}$ is the same as in G_τ.

For completeness, here is the combined scheme $G_{\tau'}$ (axiom E):

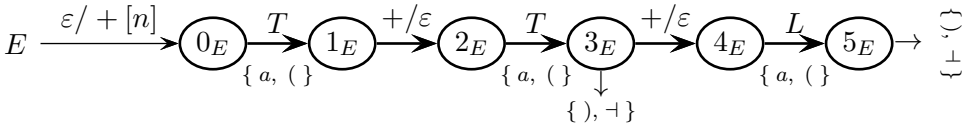
$$G_{\tau'} \begin{cases} E \to \frac{\varepsilon}{+}\, T\, \frac{\pm}{\varepsilon}\, T \\ E \to \frac{\varepsilon}{+n}\, T\, \frac{\pm}{\varepsilon}\, T\, \frac{\pm}{\varepsilon}\, L \\ L \to T\, \frac{\pm}{\varepsilon}\, L \\ L \to T \\ T \to \frac{(}{\varepsilon}\, E\, \frac{)}{\varepsilon} \\ T \to \frac{a}{a} \end{cases}$$

The source and destination trees can be overlapped accordingly.

(c) Scheme G_τ is essentially the classical infix-to-prefix translation, well-known to be deterministically computable by an $ELL(1)$ parser with write actions. For completeness, here is the whole transduction net (the call arcs are not shown):



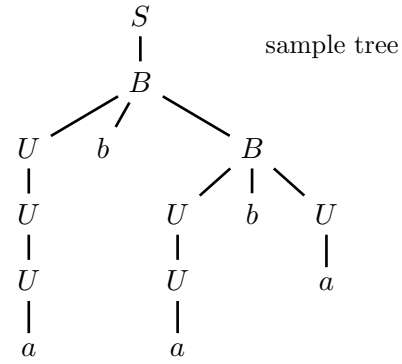
The transduction net does not have any guide set conflicts on the bifurcation states (1_E, 1_L and 0_T), and it outputs a single string per each input string. Thus the transducer is implementable as an $ELL(1)$ parser with write actions.
Instead, scheme $G_{\tau'}$ is not so. By factoring the common prefix $T + T$ of the two alternative source rules of E, we obtain this finite-state transducer:



The bifurcation state 3_E does not have any guide set conflict and the addition sign "+" is correctly output initially, as before. Yet now the placeholder n is optional, depending on whether addition will be binary or ternary (or more), which is still unknown. The transducer is $ELL(1)$ on the source, but it is unable to write the translation at the very moment when it has to be output. Said differently, the transducer should have multiple output, incompatibly with determinism. In fact, it is well-known that determinizing a finite-state transducer that should compute a function, from a string to one string, may be impossible. It is also impossible to remedy through a look-ahead window of width $k > 1$. In fact, in the transducer above, even the first term T could generate a subexpression of arbitrary length, thus delaying the discovery of the addition type, whether binary or ternary (or more), to an unlimited distance, incompatibly with a window of limited width. In conclusion, translation τ' is not deterministically computable by an $ELL(k)$ parser with write actions, for any $k \geq 1$.

2. The grammar below (axiom S), over a two-letter terminal alphabet $\{ a, b \}$, models the abstract syntax of (simplified) expressions with binary operator b and atomic operand a. The copy rule $U \to U$ allows the trees of such expressions to have non-branching paths of arbitrary length, as exemplified in the sample tree on the right:

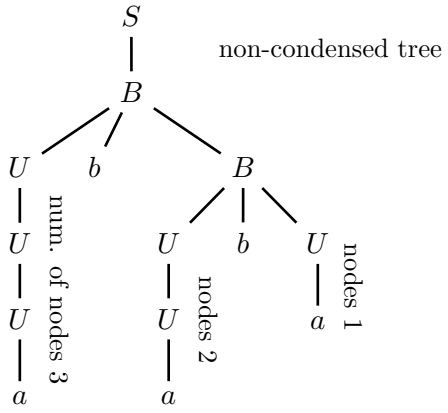
$$\begin{cases} S \to B \\ S \to U \\ U \to U \\ U \to a \\ B \to U\ b\ B \\ B \to U\ b\ U \end{cases}$$



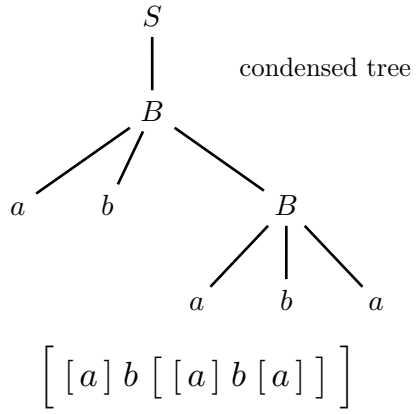
sample tree

This grammar models an *abstract* syntax, thus it is unsuitable for syntax analysis.

A *condensed* syntax tree is one where the internal nodes of a non-branching path are canceled, as shown below (related to the same sample tree as above):



non-condensed tree



condensed tree

$\Big[\ [\,a\,]\ b\ \Big[\ [\,a\,]\ b\ [\,a\,]\ \Big]\ \Big]$

linear representation of the condensed tree

Answer the following questions (use the tables / drawings / spaces on the next pages):

(a) Write an attribute grammar that computes, in the tree root, a string constituting the linear (parenthesized) representation of the *condensed* syntax tree. Use only the attribute s specified, and, to compute it, use the concatenation operator "$\cdot$". Make sure that the grammar is one-sweep and explain why.

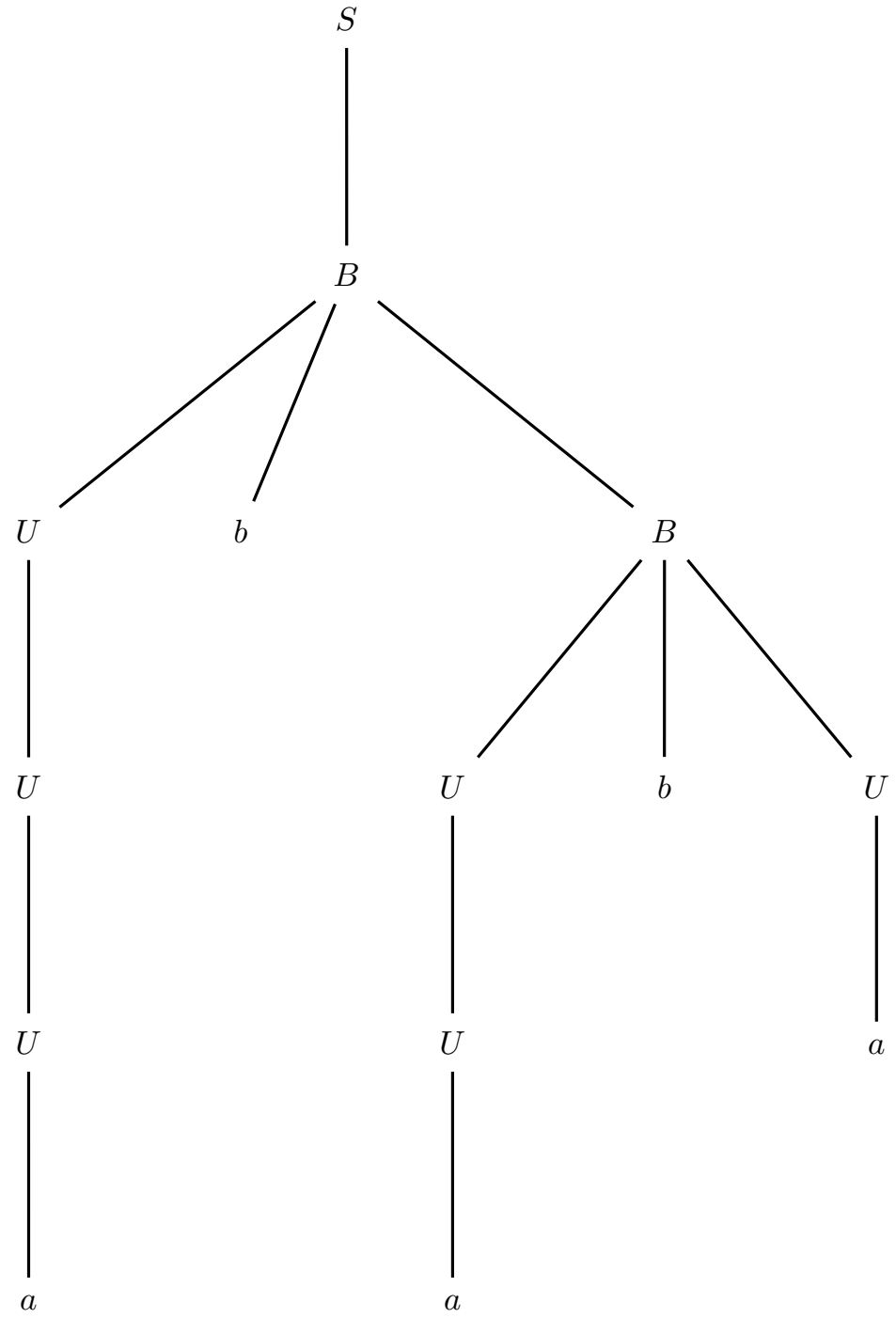
(b) Write an attribute grammar that computes, in the tree root, an *extended* linear representation of the condensed syntax tree, where an integer is inserted next to each substring "$[\,a\,]$". Such an integer indicates the number of consecutive nodes U occurring in the non-branching path immediately above the corresponding leaf node "a". The extended linear representation of the sample tree will thus be "$\big[\ [\,a\,]3\ b\ \big[\ [\,a\,]2\ b\ [\,a\,]1\ \big]\ \big]$". Use both the attributes s and n specified. To insert such integers into the extended linear representation, you may use the function $itos(i)$, which converts an integer i into its string representation.

(c) (optional) Decorate the sample tree with the values of attributes s and n.

attribute specifications – questions (a) and (b)

name	*type*	*domain*	*symbol*	*meaning*
s	left	string	S, B, U	the linear representation of the condensed (sub)tree rooted at the current node
n	left	integer ≥ 1	U	the number of consecutive nodes U in a vertical downward path starting from the current node (included); for instance, for the three branches of nodes U in the sample tree (from left to right), attribute n is equal to 3, 2 and 1

question (c) – decorate here the sample syntax tree with attributes s and n



Solution

(a) The idea is to encode the operand a between square brackets at the leaf level (rule 4), to propagate the partial linear representations upwards (rule 3), and to concatenate them whenever an operator node b is encountered (rules 5 and 6). The final result emerges on top (rules 1 and 2). This is similar to how an arithmetic expression is computed bottom-up. Here is the attribute grammar, quite simple:

#	*syntax*	*semantics* – question (a)
1:	$S_0 \to B_1$	$s_0 := s_1$
2:	$S_0 \to U_1$	$s_0 := s_1$
3:	$U_0 \to U_1$	$s_0 := s_1$
4:	$U_0 \to a$	$s_0 := [\cdot a \cdot]$
5:	$B_0 \to U_1\ b\ B_2$	$s_0 := [\cdot s_1 \cdot b \cdot s_2 \cdot]$
6:	$B_0 \to U_1\ b\ U_2$	$s_0 := [\cdot s_1 \cdot b \cdot s_2 \cdot]$

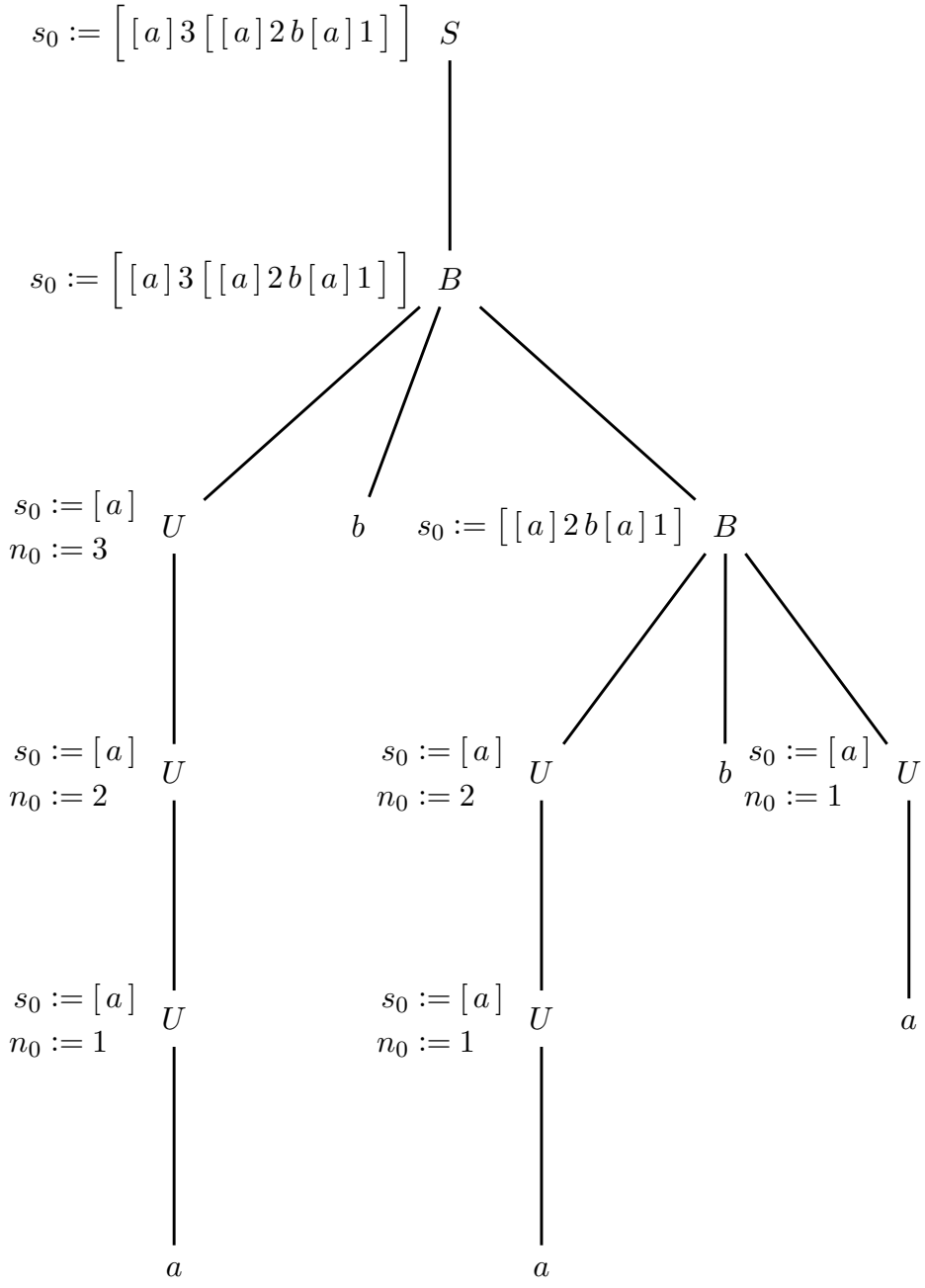
The attribute grammar does not have any circular dependence, because it uses only one attribute, thus it is correct. Furthermore, the grammar is one-sweep because it is purely synthesized.

(b) The idea is similar to the one before, but additionally the depth is computed bottom-up (rules 3 and 4) and is inserted next to the corresponding operand a whenever an operator node b is encountered (rules 5 and 6) or the root is reached (rule 2). Here is the attribute grammar, almost as simple as before:

#	*syntax*	*semantics* – question (b)
1:	$S_0 \to B_1$	$s_0 := s_1$
2:	$S_0 \to U_1$	$s_0 := s_1 \cdot \mathit{itos}\,(n_1)$
3:	$U_0 \to U_1$	$s_0 := s_1$ $n_0 := n_1 + 1$
4:	$U_0 \to a$	$s_0 := [\cdot a \cdot]$ $n_0 := 1$
5:	$B_0 \to U_1\ b\ B_2$	$s_0 := [\cdot s_1 \cdot \mathit{itos}\,(n_1) \cdot b \cdot s_2 \cdot]$
6:	$B_0 \to U_1\ b\ U_2$	$s_0 := [\cdot s_1 \cdot \mathit{itos}\,(n_1) \cdot b \cdot s_2 \cdot \mathit{itos}\,(n_2) \cdot]$

The decorated tree of question (c) provides a computation example. The attribute grammar does not have any circular dependence, because the only interaction between attributes s and n is in the rules 2, 5 and 6, and clearly it is not circular. Thus the grammar is correct. Furthermore, it is one-sweep because it is purely synthesized.

(c) Here is the tree decoration for question (b), quite immediate:

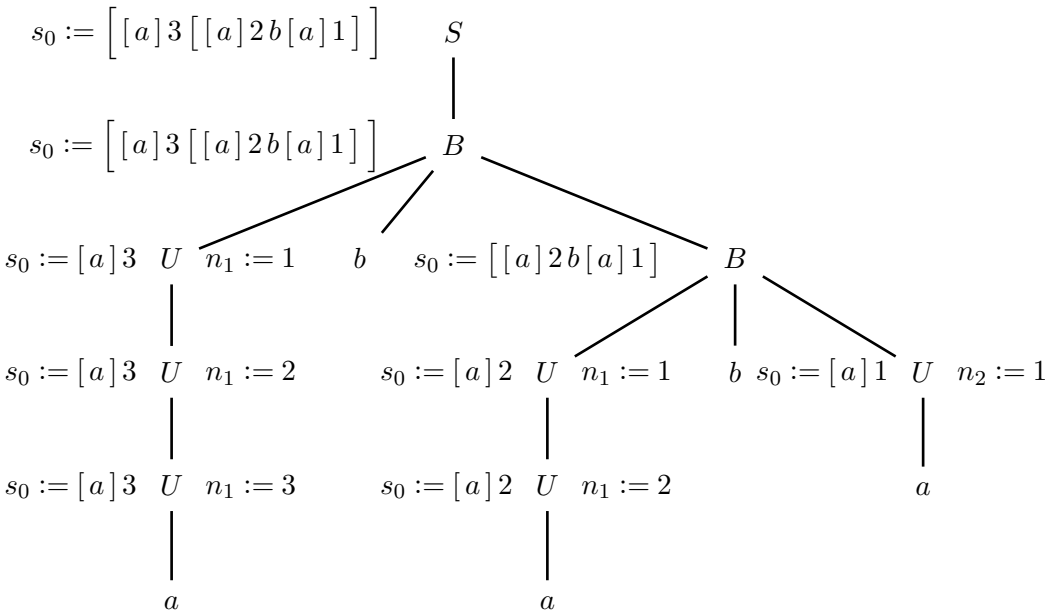


Both the left attributes s and n are initialized and propagated upwards together. The integer is inserted next to the operand when the operator b is also included. According to the usual conventions, both attributes, which are synthesized (left), are written on the left side of the node they refer to.

As a further development, notice that question (b) could be as well solved by means of a *right* attribute n, insted of left. Here is the modified solution:

#	*syntax*	*semantics* – question (b) with attribute n right
1:	$S_0 \rightarrow B_1$	$s_0 := s_1$
2:	$S_0 \rightarrow U_1$	$n_1 := 1$ $s_0 := s_1$
3:	$U_0 \rightarrow U_1$	$n_1 := n_0 + 1$ $s_0 := s_1$
4:	$U_0 \rightarrow a$	$s_0 := [\cdot a \cdot] \cdot itos\,(n_0)$
5:	$B_0 \rightarrow U_1\, b\, B_2$	$n_1 := 1$ $s_0 := [\cdot s_1 \cdot b \cdot s_2 \cdot]$
6:	$B_0 \rightarrow U_1\, b\, U_2$	$n_1,\, n_2 := 1$ $s_0 := [\cdot s_1 \cdot b \cdot s_2 \cdot]$

The difference with respect to grammar (b) is that the deph is inserted next to the related openard a in the rule (4), immediately when the operand itself is encoded. The attribute grammar does not have any circular dependence, because the only interaction between attributes s and n is in the rule 4, and clearly it is not circular. Thus the grammar is correct. Furthermore, the grammar is one-sweep, too, because the right attribute n depends only on itself, and the left attribute s depends on itself and on the right attribute n, but in the same node (rule 4). Thus the one-sweep condition is satisfied. The decorated tree becomes as follows (now attribute n, right, is written on the right of the node it refers to):



The final result emerges at the root and is the same as before, but the integer is first computed downwards, then (at the tree bottom) it is immediately inserted next to the operand, and finally it is propagated upwards encoded together with the operand.