FORMAL LANGUAGES AND COMPILERS

prof.s Luca Breveglieri and Angelo Morzenti

Exam of Thu 4 February 2016 - Part Theory

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

NAME:

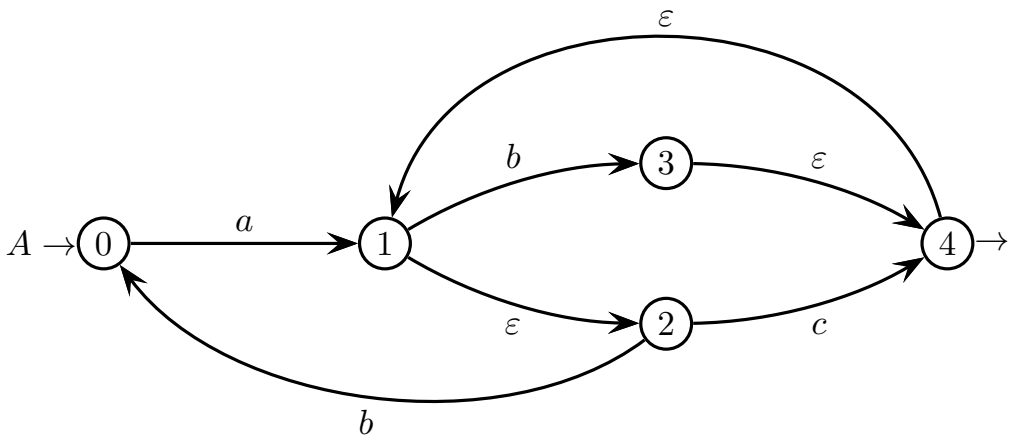
MATRICOLA: SIGNATURE:

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 finite state automaton A shown in the figure below:



Answer the following questions:

(a) In lexicographical order, write all the strings of length 3 accepted by automaton A, and for each of these strings show the computation(s) that accept(s) it.

(b) Write the right unilinear grammar G equivalent to automaton A.

(c) Use the BS method to obtain a deterministic automaton A_d equivalent to A, but without any spontaneous moves.

(d) Adopting a systematic method, determine if the deterministic automaton A_d is minimal.

(e) Using a systematic method, obtain an automaton A_c that accepts the complement of the language accepted by A.

(f) (optional) Is automaton A ambiguous? Provide an adequate, possibly informal but not tautological, justification to your answer.

Solution

(a) The four strings of length 3 accepted by automaton A are the following:

- $a\,b\,b$, computation $0 \xrightarrow{a} 1 \xrightarrow{b} 3 \xrightarrow{\varepsilon} 4 \xrightarrow{\varepsilon} 1 \xrightarrow{b} 3 \xrightarrow{\varepsilon} 4$
- $a\,b\,c$, computation $0 \xrightarrow{a} 1 \xrightarrow{b} 3 \xrightarrow{\varepsilon} 4 \xrightarrow{\varepsilon} 1 \xrightarrow{\varepsilon} 2 \xrightarrow{c} 4$
- $a\,c\,b$, computation $0 \xrightarrow{a} 1 \xrightarrow{\varepsilon} 2 \xrightarrow{c} 4 \xrightarrow{\varepsilon} 1 \xrightarrow{b} 3 \xrightarrow{\varepsilon} 4$
- $a\,c\,c$, computation $0 \xrightarrow{a} 1 \xrightarrow{\varepsilon} 2 \xrightarrow{c} 4 \xrightarrow{\varepsilon} 1 \xrightarrow{\varepsilon} 2 \xrightarrow{c} 4$

Notice that none of these four strings has two or more different accepting paths, despite automaton A is not deterministic; see also question (d).

(b) Here is the right (strictly) unilinear grammar G associated with automaton A (axiom 0), with rules that map the moves of A in a one-to-one way:

$$G \begin{cases} 0 \;\to\; a\;1 \\ 1 \;\to\; 2 \;\mid\; b\;3 \\ 2 \;\to\; b\;0 \;\mid\; c\;4 \\ 3 \;\to\; 4 \\ 4 \;\to\; \varepsilon \;\mid\; 1 \end{cases}$$

An equivalent regular expression is:

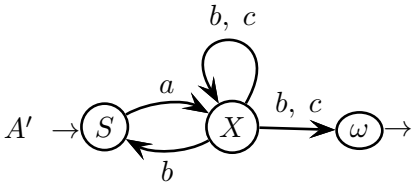
$$R = a\,\Big(\,(\,b\,a\,)^*\;(\,b\;\mid\;c\,)\,\Big)^+$$

informally obtained from A or G.

The nonterminal set of G is $V = \{\ 0,\ 1,\ 2,\ 3,\ 4\ \}$, i.e., the state set of automaton A. Grammar G has three copy rules, due to the spontaneous transitions of A. On the right, it is shown a regular expression R equivalent to grammar G.

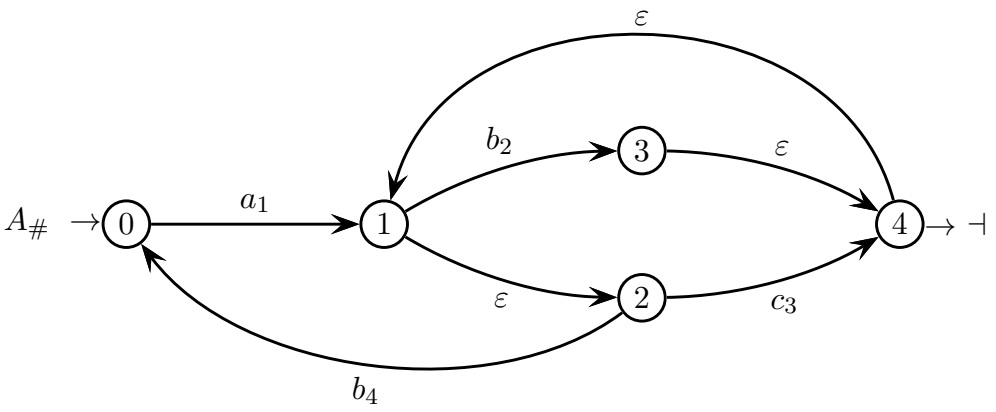
A more compact right (yet not strictly) unilinear grammar G', with fewer nonterminals (only two), but still equivalent to automaton A, can be obtained from grammar G by removing all copy rules and nullable nonterminals (axiom S):

$$G' \begin{cases} S \;\to\; a\;X \\ \\ X \;\to\; b\;S \;\mid\; b\;X \;\mid\; c\;X \;\mid\; b \;\mid\; c \end{cases}$$



The small automaton A' on the right is equivalent to grammar G', hence it is equivalent to automaton A as well, and its moves are in a one-to-one correspondence with the rules of G'. It is heavily nondeterministic, e.g., state S has three outgoing transitions with the same label b, and it has fewer states than the equivalent deterministic minimal automaton A_d found at question (c). It demonstrates that sometimes nondeterminism may help to save states.

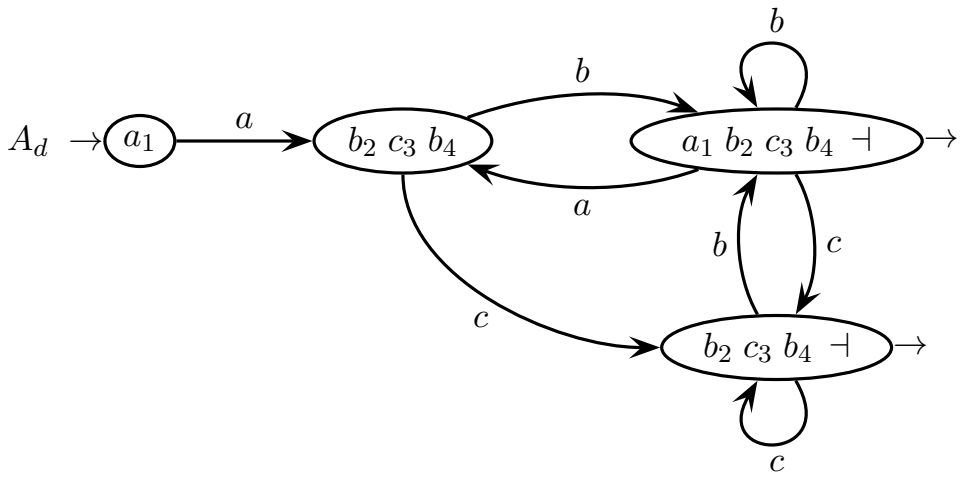
(c) The (standard) Berry-Sethi method is applied to the automaton A with spontaneous transitions. Here is the automaton $A_\#$ with distinguished transitions:



Here are the initial and follower sets of the numbered automaton $A_\#$:

initials	a_1
terminals	followers
a_1	$b_2\ c_3\ b_4$
b_2	$b_2\ c_3\ b_4\ \dashv$
c_3	$b_2\ c_3\ b_4\ \dashv$
b_4	a_1

And here is the *BS* (deterministic) automaton A_d obtained from A:



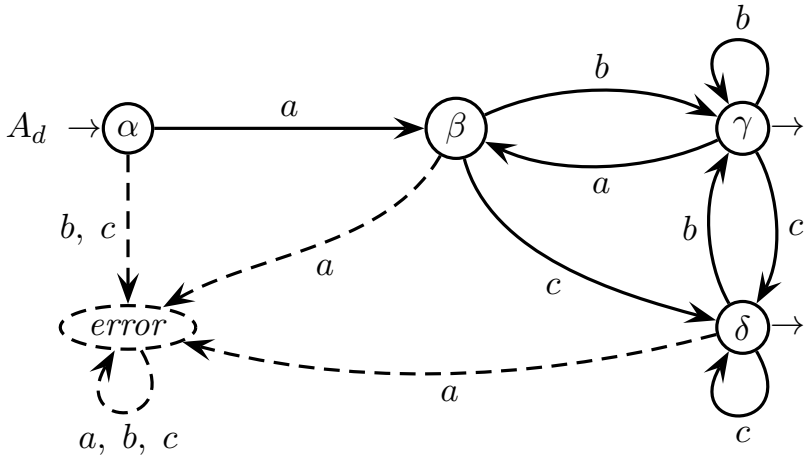
with four states, two of which are final. It is easy to check that automaton A_d accepts the four strings of length 3 that were found at point (a).

Incidentally, one may notice that the deterministic automaton A_d is local (this happens just by chance): each state has all its ingoing arcs (if any) labeled by the same input character, and no two states have the same input character. Of course, the original automaton A is not local, as it is not deterministic.

(d) The Berry-Sethi (BS) automaton A_d is in the clean form: all the states are reachable and defined, i.e., all are useful. Then, the two non-final states are distinguishable, as they have different outgoing labels (letter a), and the two final states are also distinguishable, for the same reason (letter a again). In conclusion all the states of A_d are distinguishable, hence automaton A_d is minimal.

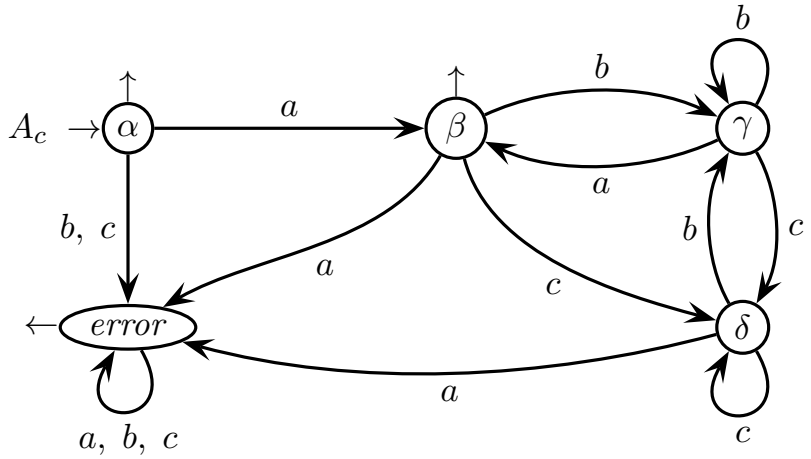
For a fully systematic check, one can draw the complete triangular table of state undistinguishability and solve it iteratively. This is left to the reader.

(e) The BS automaton A_d is deterministic by construction, but it is not in the complete form. To complement it, first it has to be naturally completed by introducing the error state and the related transitions (dashed), as follows:



where the state names have been shortened, for simplicity.

Then the complement automaton A_c is obtained from the complete form of autoomaton A_d by switching the final and non-final states, as follows:



One can check that automaton A_c rejects the four strings of length 3 of question (a). Of course, automaton A_c is deterministic and also naturally complete by construction. The reader may wish to check if it is minimal, too (not necessarily).

(f) Remember that by definition a nondeterministic finite state automaton is said to be ambiguous if there is a valid string that labels two or more different accepting paths in the automaton. Equivalently, a nondeterministic finite state automaton is ambiguous if and only if the associated right unilinear grammar is ambiguous.

Now, automaton A is not ambiguous. Here is a semi-formal proof: for every state of A where the computation can continue by following distinct transitions that eventually lead to the acceptance of a string, i.e., states 1 and 2, the strings read by distinct accepting computations originating from those transitions differ starting at least from the second symbol. Therefore, there do not exist any two distinct computations that accept the same string.

For instance, starting from state 1, if transition $1 \to 3$ is taken, then the strings are $\{\ b\ \dashv,\ b\,b\ \ldots,\ b\,c\ \ldots,\ \ldots\ \}$, while, if transition $1 \to 2$ is taken, then the strings are $\{\ b\,a\ \ldots,\ c\ \dashv,\ c\,b\ \ldots,\ \ldots\ \}$; one sees the second chars are different.

By the way, this argument also (semi-formally) proves that the right unilinear grammar G of automaton A constructed at point (b) is not ambiguous either.

A fully formal proof that the nondeterministic automaton A is not ambiguous, consists for instance of testing the pilot graph of the right unilinear grammar associated to A. Since such a grammar is purely *BNF*, the pilot graph can be drawn by simply using marked rules, else one has first to build a machine net for the grammar and then to draw its pilot. If the pilot (no matter how it is obtained) satisfies the *LR* or *ELR* condition, then the grammar is not ambiguous and consequently automaton A is not ambiguous either. One should take care of using precisely the right unilinear grammar whose rules are in one-to-one correspondence with the moves of automaton A, i.e., grammar G of question (b); other equivalent grammars, though unilinear as well, may not have the same ambiguity as automaton A, therefore analyzing them may not be fully representative of the behaviour of A. This formal proof is left to the reader.

2 Free Grammars and Pushdown Automata 20%

1. Consider the Dyck language with two parenthesis types, namely $a\ b$ (open closed) and $c\ d$ (open closed), i.e., over the alphabet $\Sigma = \{\, a,\ b,\ c,\ d \,\}$, generated by the well known non-ambiguous *BNF* grammar G shown below (axiom S):

$$G \begin{cases} S & \rightarrow & a\ S\ b\ S \\ S & \rightarrow & c\ S\ d\ S \\ S & \rightarrow & \varepsilon \end{cases}$$

   Answer the following questions:

   (a) Write a grammar G_1, *BNF* and non-ambiguous, that generates language $L\,(G)$ minus all the Dyck strings that contain one or more pairs of adjacent letters $a\,c$.

   (b) (optional) Write a grammar G_1, *BNF* and non-ambiguous, that generates language $L\,(G)$ minus all the Dyck strings that contain one or more pairs of adjacent letters $b\,d$.

Solution

(a) A letter pair $a\,c$ can be generated only by a derivation step with a rule $S \to a\ S\ b\ S$ whose leftmost occurrence of nonterminal S in the right part is immediately expanded by a rule $S \to c\ S\ d\ S$, i.e., a derivation like $S \overset{S \to a\ S\ b\ S}{\Longrightarrow} a\ \underline{S}\ b\ S \overset{S \to c\ S\ d\ S}{\Longrightarrow} a\ c\ S\ d\ S\ b\ S$. One needs to exclude such a derivation case, while all the others are still permitted.

To do so, it suffices to distinguish the nonterminal S that is immediately preceded by terminal a - change the name of such an instance of S into A instead - so as to disallow this new nonterminal A to generate a string that starts by letter c. Here is a version of such a grammar G_1 (axiom S):

$$G_1 \left\{ \begin{array}{lcll} S & \to & a\ A\ b\ S & // \ A \text{ may not generate an initial letter } c \\ S & \to & c\ S\ d\ S & \\ S & \to & \varepsilon & \\ \hline A & \to & a\ A\ b\ S & // \ A \text{ is unable to generate an initial letter } c \\ A & \to & \varepsilon & \end{array} \right.$$

Grammar G_1 is *BNF* and non-ambiguous by construction, as it is an obvious restriction of a non-ambiguous one. A minor observation: it has rules with a repeated right part, and it can be made slightly less redundant by categorization, that is, by introducing a copy rule, like in this version G'_1 (axiom S):

$$G'_1 \left\{ \begin{array}{lcl} S & \to & A \mid c\ S\ d\ S \\ \hline A & \to & a\ A\ b\ S \mid \varepsilon \end{array} \right.$$

with grouped alternative rules. Different solutions may exist, of course.

(b) To exclude a letter pair $b\,d$, one only needs to disallow a derivation sequence like $S \overset{S \to c\ S\ d\ S}{\Longrightarrow} c\ \underline{S}\ d\ S \overset{S \to a\ S\ b\ S}{\Longrightarrow} c\ a\ S\ b\ \underline{S}\ d\ S \overset{S \to \varepsilon}{\Longrightarrow} c\ a\ S\ b\ \varepsilon\ d\ S$, and the longer ones (without rules) like $S \Rightarrow c\ \underline{S}\ d\ S \Rightarrow c\ x\ S\ y\ \underline{S}\ d\ S \Rightarrow c\ x\ S\ y\ a\ S\ b\ \underline{S}\ d\ S \Rightarrow c\ x\ S\ y\ a\ S\ b\ \varepsilon\ d\ S$ (letter pair $x\,y$ may be $a\,b$ or $c\,d$), and so on.

Now it suffices to distinguish the nonterminal S that is immediately followed by terminal d - change the name of such an instance of S into D instead - so as to disallow this new nonterminal D to generate a string that ends by letter b. Here is a version of such a grammar G_2 (axiom S):

$$G_2 \left\{ \begin{array}{lcll} S & \to & a\ S\ b\ S & \\ S & \to & c\ D\ d\ S & // \ D \text{ may not generate a final letter } b \\ S & \to & c\ d\ S & // \text{ rule to nullify the occurrence of } D \\ S & \to & \varepsilon & \\ \hline D & \to & a\ S\ b\ D & // \ D \text{ is unable to generate a final letter } b \\ D & \to & c\ D\ d\ D & // \text{ idem, but for both occurrences of } D \\ D & \to & c\ d\ D & // \text{ rule to nullify the left occurrence of } D \\ D & \to & c\ D\ d & // \text{ idem, but for the right occurrence} \\ D & \to & c\ d & // \text{ idem, but for both occurrences} \end{array} \right.$$

Of course, a key point is that nonterminal D may not generate the empty string, for else it could generate strings that end by letter b. This causes a number (though limited) of rule variants to show up, wherein the occurrences of nonterminal D are canceled in all possible ways independently of one another; it is similar to the construction for the elimination of a nullable nonterminal.

Grammar G_2 is *BNF* and non-ambiguous by construction, being essentially made of more or less restricted Dyck rules. It works fine, yet making nonterminal D non-nullable causes the rules to proliferate. One may wonder whether one could eliminate or reduce them. Remember that the standard Dyck rule comes in two equivalent forms: right (as before in G_2) or left recursive, i.e., $S \to S\ x\ S\ y$. The latter form is better suited for the current purpose (grammar G_3 axiom S):

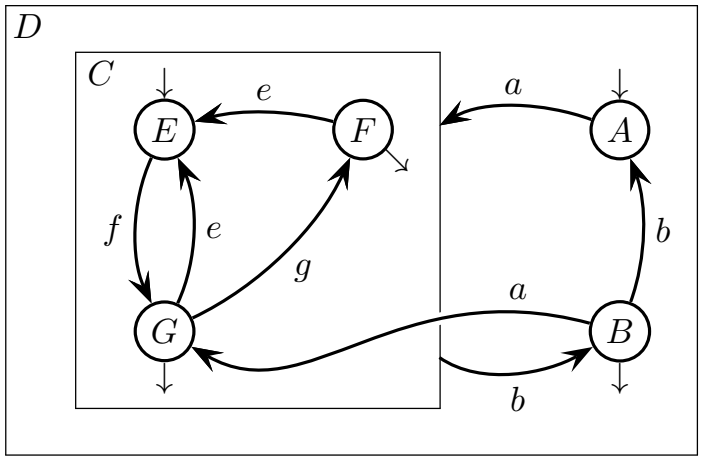
$$G_3 \left\{ \begin{array}{lcll} S & \to & S\ a\ S\ b & \\ S & \to & S\ c\ D\ d & // \ D \text{ may not generate a final letter } b \\ S & \to & \varepsilon & \\ \hline D & \to & S\ c\ D\ d & // \ D \text{ is unable to generate a final letter } b \\ D & \to & \varepsilon & \end{array} \right.$$

Grammar G_3 is left-recursive and is equivalent to grammar G_2 (right-recursive instead), as clearly it can generate all Dyck strings except those with a letter pair $b\,d$, since nonterminal D never generates strings that end by letter b. It is also non-ambiguous, for the same reasons as of G_2, and by categorization it could be freed from repeated right parts, like done for grammar G'_1 (axiom S):

$$G'_3 \left\{ \begin{array}{lcl} S & \to & S\ a\ S\ b\ \mid\ D \\ \hline D & \to & S\ c\ D\ d\ \mid \varepsilon \end{array} \right.$$

An *a-posteriori* observation: grammars G_1 and G_3 of questions (a) and (b), respectively, are structurally identical ! To see how this shows up, just switch nonterminals A and D, and similarly switch the parenthesis structures $a\,b$ and $c\,d$. Here is the reason why this happens: the two questions posed in (a) and (b) are conceptually the same, though with open and closed brackets switched: a vs b, c vs d, hence pair $a\,c$ vs pair $b\,d$; but this corresponds to switch grammar recursion: right vs left. To visualize, first read a Dyck string from right to left: closed and open brackets play each other's role; then, similarly read the right part of a Dyck rule of G (the grammar shown in the text) from right to left: right recursion turns into left. Of course, such a subtle structural relation is not immediate to mentally grasp. Other, totally different solutions may exist.

2. A state diagram, inspired to the notion of *Statecharts*, is a generalization of a finite state automaton, where any state may be refined into a lower-level state diagram. This refinement operation can be iterated at any depth level. The figure below shows the graphical representation of a state diagram D with three states, where state C is refined into a lower-lever state diagram with states E, F and G.



The text description of the state diagram in the figure above is the following:

```
stateDiagram D is

   state A, B, C;
   initial A;
   final B;
   transition A to C label a, B to A label b;
   transition C to B label b, B to C.G label a;

   stateDiagram C is

      state F, G, E;
      initial E;
      final G;
      final F;
      transition E to G label f, G to E label e, F to E label e,
      G to F label g;

   end stateDiagram C;

end stateDiagram D;
```

A state diagram must satisfy the following constraints:

- There is exactly one clause for declaring all the atomic states, i.e., the states not refined into a state (sub)diagram; notice that a diagram may include one or more atomic states, and that it may even be empty.
- There is exactly one initial state, declared in only one clause.
- There are one or more final states, declared in as many clauses (one per clause).
- A transition may connect a state of a diagram to a state at the same level or to one in a lower-level diagram, at any depth level, for instance as in:

  transition Q to P.R.S label m

  The syntax that generates the states must define the operator "." to be left-associative, so that a state pathname like "P.R.S" is parsed as "(P.R).S".
- The relative position of the various clauses within the definition of a diagram is defined by the provided example.

Write an extended (*EBNF*) grammar, not ambiguous, that generates a non-empty sequence of state diagram definitions according to the rules described above.

Solution

Here is the requested grammar (axiom PROG):

$$
\begin{array}{rcl}
\langle \mathsf{PROG} \rangle & \rightarrow & \big(\ \langle \mathsf{DIAG} \rangle \ \text{`;'} \ \big)^{+} \\
\hline
\langle \mathsf{DIAG} \rangle & \rightarrow & \mathsf{stateDiagram} \ \langle \mathsf{ID} \rangle \ \mathsf{is} \\
 & & \quad \big[\ \langle \mathsf{ST\_LST} \rangle \ \text{`;'} \ \big] \\
 & & \quad \big[\ \langle \mathsf{INITIAL} \rangle \ \text{`;'} \ \big] \\
 & & \quad \big(\ \langle \mathsf{FINAL} \rangle \ \text{`;'} \ \big)^{*} \\
 & & \quad \big(\ \langle \mathsf{TR\_LST} \rangle \ \text{`;'} \ \big)^{*} \\
 & & \quad \big[\ \langle \mathsf{PROG} \rangle \ \big] \\
 & & \mathsf{end\ stateDiagram} \ \langle \mathsf{ID} \rangle \ \text{`;'} \\
\hline
\langle \mathsf{ST\_LST} \rangle & \rightarrow & \mathsf{state} \ \langle \mathsf{ID} \rangle \ \big(\ \text{`,'} \ \langle \mathsf{ID} \rangle \ \big)^{*} \\
\langle \mathsf{INITIAL} \rangle & \rightarrow & \mathsf{initial} \ \langle \mathsf{ID} \rangle \\
\langle \mathsf{FINAL} \rangle & \rightarrow & \mathsf{final} \ \langle \mathsf{ID} \rangle \\
\langle \mathsf{TR\_LST} \rangle & \rightarrow & \mathsf{transition} \ \langle \mathsf{TR\_DEF} \rangle \ \big(\ \text{`,'} \ \langle \mathsf{TR\_DEF} \rangle \ \big)^{*} \\
\hline
\langle \mathsf{TR\_DEF} \rangle & \rightarrow & \langle \mathsf{ID} \rangle \ \mathsf{to} \ \langle \mathsf{PNAME} \rangle \ \mathsf{label} \ \langle \mathsf{ID} \rangle \\
\langle \mathsf{PNAME} \rangle & \rightarrow & \langle \mathsf{PNAME} \rangle \ \text{`.'} \ \langle \mathsf{ID} \rangle \ \mid \ \langle \mathsf{ID} \rangle \quad \text{// left recursive !} \\
\hline
\langle \mathsf{ID} \rangle & \rightarrow & \text{as usual} \ldots
\end{array}
$$

Square brackets denote optionality, as usual. Identifiers could be modeled in various ways, e.g., as in the C language. Notice that as a state (sub)diagram may be empty, all the clauses (state, initial, final and transition) have to be optional. The relative ordering of the clauses is the same as the given samples suggest.

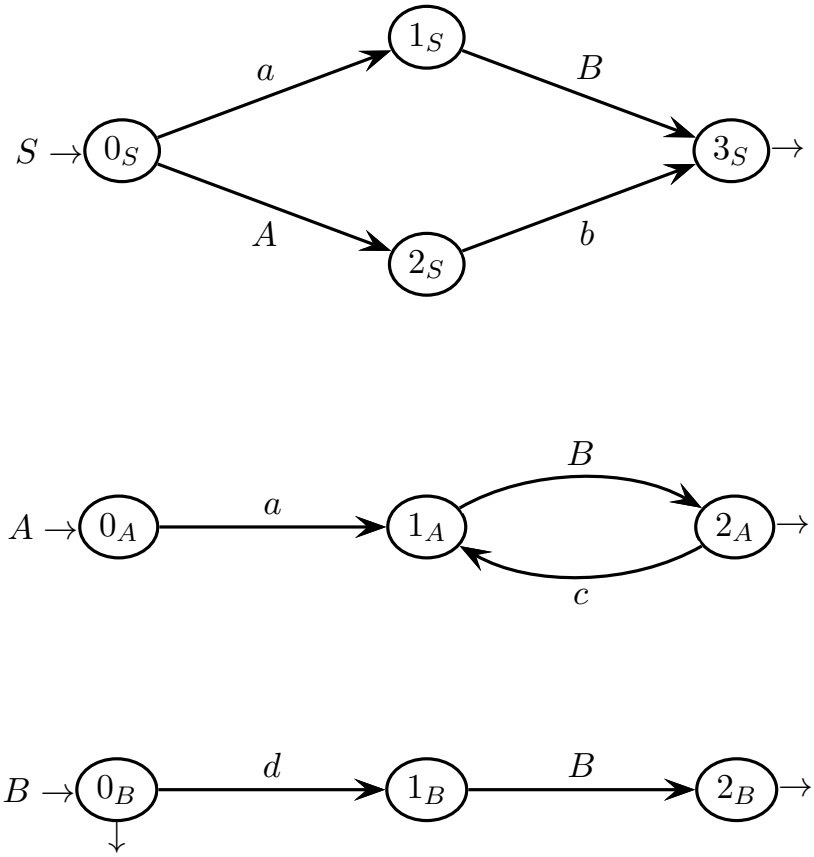
The grammar is simple and generally uses *EBNF* rules. Since however a state pathname PNAME is required to be left-associative, the rule that generates it has to be left-recursive. Though lexically a pathname is a list of identifiers separated by dots and consequently it could be generated iteratively, the left-associativity requirement prevents the usage of an *EBNF* rule with a star or cross operator, whereas the rest of the grammar is mostly *EBNF*. The axiom PROG is also used recursively to include an optional list of inner subdiagrams.

The grammar rules are layered from the higher level ones, to help the reader. Being made of well known non-ambiguous substructures (lists, etc), the grammar is not ambiguous either. One may even check that it is of type $ELR\,(1)$, hence that it is deterministic and consequently not ambiguous either. Clearly anyway, it is not of type $ELL\,(k)$, for any $k \geq 1$, as it is left-recursive, and it has to be so as said before.

Of course different solutions may exist, of type more or less *EBNF*.

3 Syntax Analysis and Parsing Methodologies 20%

1. Consider the following grammar G, represented as a machine net (axiom S):



Answer the following questions:

(a) Draw the complete *ELR* pilot graph of grammar G, say if grammar G is of type $ELR\,(1)$ and justify your answer.

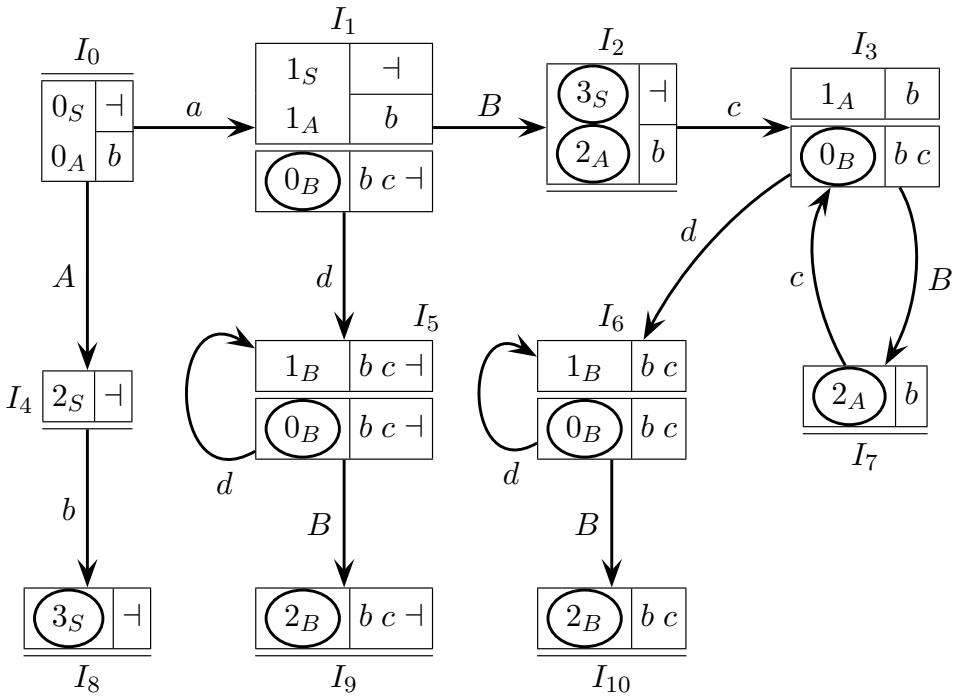
(b) Find all the guide sets with $k = 1$ on the shift arcs, call arcs and exit arrows of the machine net of grammar G, say if grammar G is of type $ELL\,(1)$ and justify your answer. Please use the net drawing above to annotate the guide sets.

(c) In the case your answer to point (b) is negative, say if grammar G is of type $ELL\,(k)$ for some $k > 1$ and shortly justify your answer.

(d) (optional) In the case your answer to point (c) is negative, shortly say if there is a grammar G' that is equivalent to G and is of type $ELL\,(k)$ for some $k \geq 1$, and shortly justify your answer.

Solution

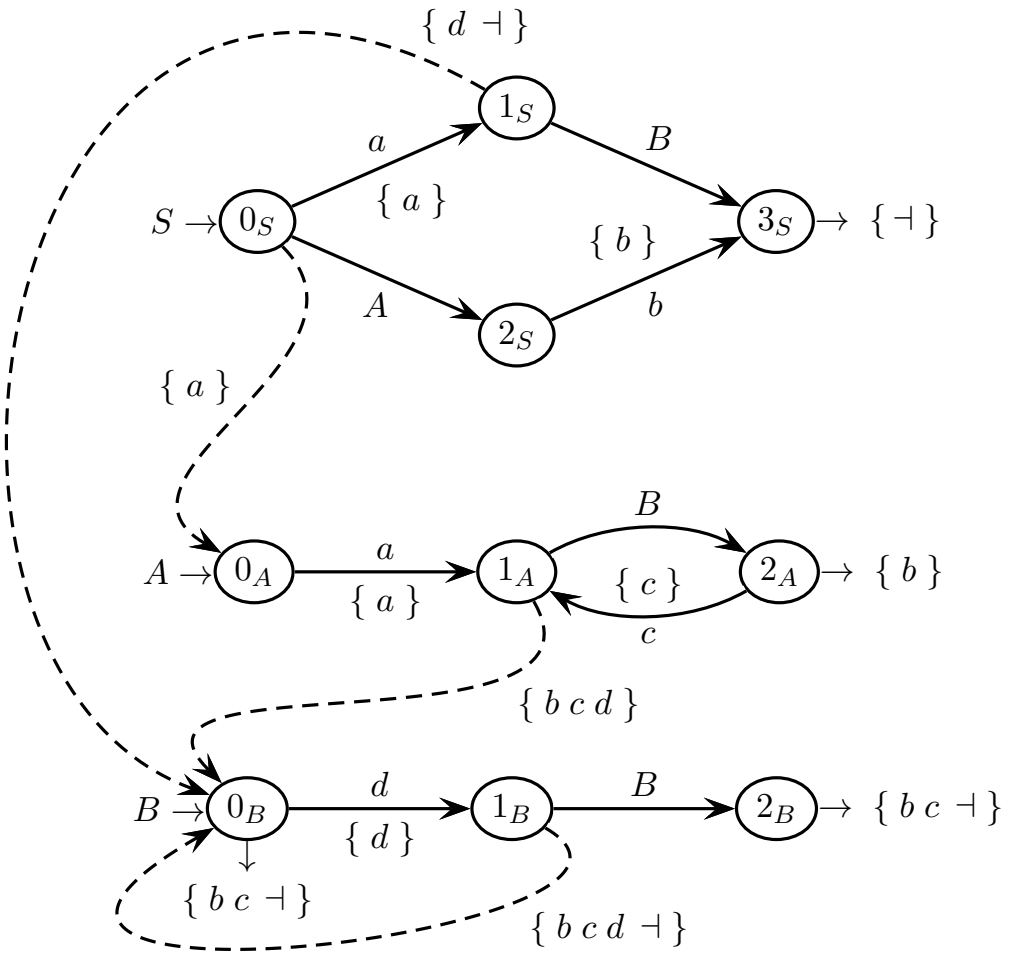
(a) The pilot graph of grammar G has 11 m-states and no conflicts of any type (shift-reduce, reduce-reduce or convergence). Here it is:



Concerning the conflicts: shift-reduce is excluded as none of the final net states (encircled) in any m-state shares a look-ahead terminal with a terminal shift transition from the m-state; reduce-reduce is excluded as it might only occur in the m-state I_2 with two final net states, but their look-ahead sets are disjoint; and convergence is excluded as the only two multiple (actually double) transitions, namely $I_0 \to I_1 \to I_2$, are not convergent to one net state (all the other transitions are single). Therefore grammar G is of type $ELR\,(1)$.

Notice that the pilot graph does not have the STP (single transition) property, as the bases of m-states I_1 and I_2 contain two candidates (items). Notice also the structural similarity between the m-state pairs I_5, I_9 and I_6, I_{10}. The only difference between them is the look-ahead character $\dashv$, the presence or absence of which reflects the different invocation points of machine B: in the m-state I_1, machine B is invoked from inside both machine S (axiomatic - so including also the terminator $\dashv$ in the look-ahead) and machine A; while in the m-state I_3, it is invoked only from inside machine A.

(b) Here are the guide sets of grammar G, annotated on the grammar *PCFG*, i.e., the machine net of G extended with call arcs (dashed):



The only overlap case of outgoing guide sets occurs at net state 0_S, due to terminal a. Anyway, this suffices to cause the *PCFG* not to be deterministic and consequently grammar G not to be of type *ELL* (1). Of course, this was also clear from question (a), as the pilot does not have the *STP* property. By the way, notice that the guide set overlap occurs on the *PCFG* transitions related to the pilot double transitions $I_0 \to I_1$ and $I_1 \to I_2$ that violate the *STP*.

For the guide sets, one needs the prospect sets. The prospect set of final state 3_S includes only the terminator $\dashv$, because nonterminal S is the axiom and it is not recursively called anywhere in the net. The prospect set of final state 2_A includes only terminal b, because it immediately follows nonterminal A in the only net point where machine A is called. The prospect set of final state 2_B includes those of states 3_S and 2_A, because they immediately follow the invocations of B in the machines S and A, respectively, and it includes also terminal c, because it immediately follows the invocation of B in the machine A. Of course, since in this case the pilot graph is already available, the prospect sets could be quickly obtained also from the look-ahead sets of the net final states in the pilot m-states.

The guide sets on the shift arcs are immediate. Those on the exit arrows coincide with the prospect sets. Those on the call arcs need more attention, as follows.

To start, the guide set of call arc $1_B \dashrightarrow 0_B$ includes terminals b, c and the terminator $\dashv$, for the reason that nonterminal B is nullable and that the nonterminal shift arc $1_B \xrightarrow{B} 2_B$ goes to final state 2_B, the prospect set of which is $\{\ b,\ c,\ \dashv\ \}$. Beware the reason is NOT that call arc $1_B \dashrightarrow 0_B$ is followed by the exit arrow from 0_B, the prospect set of which is also $\{\ b,\ c,\ \dashv\ \}$! Terminal d is in this guide set for the simple reason that it is the initial of nonterminal B.

A similar observation applies to call arc $1_A \dashrightarrow 0_B$: it would be wrong to include the terminator $\dashv$ into the guide set of this call arc, and to justify the inclusion because the terminator appears on the exit arrow of state 0_B, which follows the call arc ! In fact, this guide set derives instead from uniting only these three elements: the prospect set contents b of final state 2_A, because nonterminal B is nullable; the guide set contents c of shift arc $2_A \xrightarrow{c} 1_A$, again because nonterminal B is nullable; and the initial terminal d of nonterminal B.

To conclude, the same observation applies also to call arc $0_S \dashrightarrow 0_B$. Here only the two following elements contribute to the guide set: the prospect set contents $\dashv$ of final state 3_S, because nonterminal B is nullable; and the initial terminal d of nonterminal B. In comparison, the guide set of call arc $0_S \dashrightarrow 0_A$ is obvious.

(c) Since clearly it holds $L(B) = d^*$ and $L(A) \cap a\, L(B) = a\, d^*$ (see also question (d)), both the guide sets of length $k \geq 2$ on the arcs outgoing from net state 0_S include all the strings $a\, d^{k-1}$ of length increasing with k (they also include other string types), hence grammar G is not of type $ELL(k)$ either, for any $k > 1$.

(d) It is not difficult to see that language $L(G)$ is regular, hence also of type $ELL(1)$ for some suitably chosen other grammar equivalent to grammar G. In fact, first:

$$L(B) = d\, L(B) \ \mid\ \varepsilon = d^*$$

by the Arden identity. Then:

$$L(A) = a\, L(B) \left(c\, L(B) \right)^* = a\, d^* \left(c\, d^* \right)^*$$

by substitution. Last:

$$L(S) = a\, L(B) \ \mid\ L(A)\, b = a\, d^* \ \mid\ a\, d^* \left(c\, d^* \right)^* b$$

again by substitution. More compactly, by factoring the sub-expression $a\, d^*$:

$$L(S) = a\, d^* \left(\varepsilon \ \mid\ \left(c\, d^* \right)^* b \right)$$

or instead by factoring only letter a and then by flattening the nested star subexpression, since it holds $d^* \ (\, c\, d^*\,)^* = (\, c \ \mid\ d\,)^*$:

$$L(S) = a \left(d^* \ \mid\ d^* \left(c\, d^* \right)^* b \right) = a \left(d^* \ \mid\ \left(c \ \mid\ d \right)^* b \right)$$

Therefore language $L(G) = L(S)$ is regular. Consequently there is a deterministic finite state automaton that recognizes language $L(G)$, and it is well known that the associated right unilinear grammar is of type $LL(1)$. Of course, there may be other $ELL(k)$ grammars, of different kinds, equivalent to grammar G.

4 Language Translation and Semantic Analysis 20%

1. Consider the translations τ_1 and τ_2, defined respectively on the following source languages L_{s1} and L_{s2}:

$$L_{s1} = \{\ a^n\, b^n\, c\ \mid\ n \geq 1\ \} \cup \{\ a^n\, b^n\, d\ \mid\ n \geq 1\ \}$$

$$L_{s2} = \{\ (a\, b)^n\, c\ \mid\ n \geq 1\ \} \cup \{\ (a\, b)^n\, d\ \mid\ n \geq 1\ \}$$

Translations τ_1 and τ_2 work as follows, for any $n \geq 1$:

$$\begin{array}{lcllcl} \tau_1\left(a^n\, b^n\, c\right) & = & c^n\, b^n\, a & \tau_2\left((a\, b)^n\, c\right) & = & (c\, b)^n\, a \\ \tau_1\left(a^n\, b^n\, d\right) & = & a^n\, d^n\, b & \tau_2\left((a\, b)^n\, d\right) & = & (a\, d)^n\, b \end{array}$$

Answer the following questions:

(a) Define translation τ_1 by means of the simplest possible translation scheme, that is, either a regular translation expression or a translation grammar.

(b) Define translation τ_2 by means of the simplest possible translation scheme, that is, either a regular translation expression or a translation grammar.

(c) (optional) Determine if the translation τ_2 can be *computed* by some of the *deterministic* transducers, i.e., deterministic I/O automata, that have been studied in the course, and adequately justify your answer.

Solution

(a) Translation τ_1 is defined by the following translation grammar G_1 (axiom S):

$$G_1 \left\{ \begin{array}{lcl} S & \rightarrow & S_1 \mid S_2 \\ S_1 & \rightarrow & \frac{a}{c}\ X_1\ \frac{b}{b}\ \frac{c}{a} \\ S_2 & \rightarrow & \frac{a}{a}\ X_2\ \frac{b}{d}\ \frac{d}{b} \\ X_1 & \rightarrow & \frac{a}{c}\ X_1\ \frac{b}{b} \mid \frac{\varepsilon}{\varepsilon} \\ X_2 & \rightarrow & \frac{a}{a}\ X_2\ \frac{b}{d} \mid \frac{\varepsilon}{\varepsilon} \end{array} \right.$$

Grammar G_1 is evident. Clearly translation τ_1 cannot be defined by means of a regular translation expression, as the source language L_{s1} is not regular.

(b) Translation τ_2 is defined by the following regular translation expression R_2:

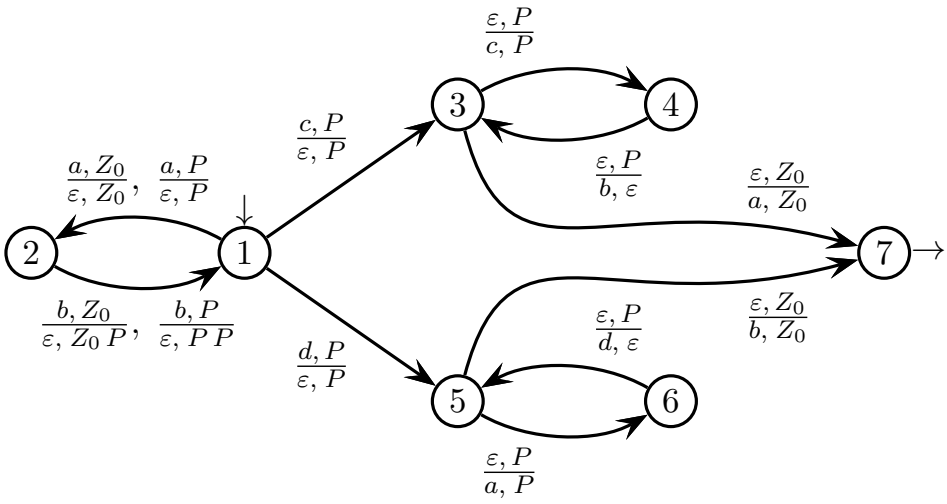
$$R_2 = \left(\left(\frac{a}{c}\ \frac{b}{b} \right)^+\ \frac{c}{a} \right) \cup \left(\left(\frac{a}{a}\ \frac{b}{d} \right)^+\ \frac{d}{b} \right)$$

Expression R_2 is evident. Of course, it is also possible to write a right linear translation grammar G_2, equivalent to the rational translation expression R_2.

Alternatively, translation τ_2 could defined by means of a nondeterministic finite state transducer (an I/O-automaton) or a deterministic $2I$-automaton (a two-tape automaton). This is left to the reader (see also question (c)).

(c) Translation τ_2 cannot be computed by a deterministic finite state translator (an I/O-automaton). In fact such a translation cannot be performed in real-time, as it depends on the last (rightmost) character of the input string.

Translation τ_2 can be however computed by a simple deterministic pushdown transducer, which behaves as follows: first it pushes all the $n \geq 1$ input character pairs $a\,b$ onto the stack and encodes them by as many stack symbols P; then, soon after finding the final input character c or d, it pops all the symbols P from the stack and outputs as many output character pairs $c\,b$ or $a\,d$, respectively; and last it outputs the final character a or b, respectively. This machine is clearly deterministic and its state-transition graph is not difficult to design. Here it is:



This pushdown translator works as described and recognizes by final state (7). Notice that the stack will be already empty in the non-final states 3 and 5, when the last pair $c\,b$ or $a\,d$ is output; termination however occurs in the final state 7. It is easy to design a similarly deterministic translator that terminates by empty stack: one need only to initially push a stack symbol X and pop it at the end.

A different approach consists of modelling translation τ_2 by means of a translation grammar (or a syntactic scheme), instead of a rational translation expression as done before, and then of checking if such a grammar is deterministic *ELL* or *ELR*. Since it has been ascertained before that translation τ_2 is deterministic pushdown, a deterministic *ELL* or *ELR* translation grammar (or scheme) should exist; it may not be so immediate to design, anyway. This is left to the reader.

2. Consider the Dyck language with two parenthesis types, namely a b (open closed) and c d (open closed), generated by the known syntactic support below (axiom S):

$$\begin{cases} 0: & S & \to & D \\ 1: & D & \to & a\ D\ b\ D \\ 2: & D & \to & c\ D\ d\ D \\ 3: & D & \to & \varepsilon \end{cases}$$

One wishes to analyze a syntax tree by means of attribute grammars, to check if the Dyck string contains certain pairs of adjacent letters, and to compute and output the nesting depth of the pair, if any, that occurs leftmost in the string, or to output 0 if there is no such pair. To answer the questions below, one has to use (partly or totally) the attributes already prepared on the next page.

Answer the following questions:

(a) Write an attribute grammar G_1 that checks if the Dyck string contains an adjacent letter pair of type $a\,c$.

(b) Write an attribute grammar G_2 that checks if the Dyck string contains an adjacent letter pair of type $b\,d$.

(c) (optional) Reconsider grammar G_1 and extend it, in such a way that it outputs (at the tree root) the nesting depth (always ≥ 1) of the leftmost occurrence of the pair $a\,c$ (more precisely the depth of letter c in the pair), if the Dyck string has some, or else that outputs 0. Samples:

- $a\,c\,d\,b$, yes (one pair), depth 1
- $a\,b\,a\,c\,d\,b$, yes (one pair), depth 1
- $a\,a\,b\,a\,c\,d\,b\,b$, yes (one pair), depth 2
- $a\,c\,d\,a\,a\,c\,d\,b\,b$, yes (two pairs), depth 1
- $a\,b\,c\,d$, none, depth 0

Furthermore, say if the so-extended attribute grammar is of type one-sweep and justify your answer. To answer question (c), you can (and actually have to) define more attributes, for dealing with the nesting depth. Complete the attribute list already given on the next page.

---

attributes assigned to be used (complete the specification if necessary)

type	*name*	*domain*	*(non)term.*	*meaning*
left	b	boolean	D	true if and only if nonterminal D generates a string that ends with letter b
left	c	boolean	D	true if and only if nonterminal D generates a string that starts with letter c
left	e	boolean	D	true if and only if nonterminal D generates the empty string
left	ac	boolean	S, D	true at the tree root if and only if letter pair $a\,c$ occurs in the Dyck (sub)string generated by S (or by D)
left	bd	boolean	S, D	true at the tree root if and only if letter pair $b\,d$ occurs in the Dyck (sub)string generated by S (or by D)

attributes to be added, if any (complete the specification)

type	*name*	*domain*	*(non)term.*	*meaning*

#	*syntax*	*semantics* - question (a)
0:	$S_0 \rightarrow D_1$	
1:	$D_0 \rightarrow a\ D_1\ b\ D_2$	
2:	$D_0 \rightarrow c\ D_1\ d\ D_2$	
3:	$D_0 \rightarrow \varepsilon$	

#	*syntax*	*semantics* - question (b)

0: $S_0 \rightarrow D_1$

1: $D_0 \rightarrow a\ D_1\ b\ D_2$

2: $D_0 \rightarrow c\ D_1\ d\ D_2$

3: $D_0 \rightarrow \varepsilon$

#	*syntax*	*semantics* - question (c)

0: $S_0 \rightarrow D_1$

1: $D_0 \rightarrow a\ D_1\ b\ D_2$

2: $D_0 \rightarrow c\ D_1\ d\ D_2$

3: $D_0 \rightarrow \varepsilon$

Solution

(a) Here is the solution, purely synthesized, where attributes c and ac suffice:

#	*syntax*	*semantics* - question (a)
0:	$S_0 \rightarrow D_1$	$ac_0 = ac_1$
1:	$D_0 \rightarrow a\ D_1\ b\ D_2$	$c_0 = false$ $ac_0 = c_1\ \vee\ ac_1\ \vee\ ac_2$
2:	$D_0 \rightarrow c\ D_1\ d\ D_2$	$c_0 = true$ $ac_0 = ac_1\ \vee\ ac_2$
3:	$D_0 \rightarrow \varepsilon$	$c_0 = false$ $ac_0 = false$

Remember that a connective "$\vee$" means logical sum (disjunction). Basically, see rule 1, a letter pair $a\,c$ occurs in the Dyck (sub)string generated by D_0 if nonterminal D_1 immediately generates letter c, or if a pair $a\,c$ already occurs in the Dyck substrings generated by D_1 or D_2. Similarly for rule 2, except that in this case an immediate pair $a\,c$ may not show up, as the rule does not contain letter a. The other semantic functions are obvious.

(b) Here is the solution, purely synthesized, where attributes b, e and bd suffice:

#	*syntax*	*semantics* - question (b)
0:	$S_0 \rightarrow D_1$	$bd_0 = bd_1$
1:	$D_0 \rightarrow a\ D_1\ b\ D_2$	$b_0 = b_2\ \vee\ e_2$ $e_0 = false$ $bd_0 = bd_1\ \vee\ bd_2$
2:	$D_0 \rightarrow c\ D_1\ d\ D_2$	$b_0 = b_2$ $e_0 = false$ $bd_0 = b_1\ \vee\ bd_1\ \vee\ bd_2$
3:	$D_0 \rightarrow \varepsilon$	$b_0 = false$ $e_0 = true$ $bd_0 = false$

Attribute grammar G_2 is quite similar to grammar G_1, but computing attribute b is (only) slightly more difficult than computing attribute c. Basically, see rule 2, a letter pair $b\,d$ occurs in the Dyck (sub)string generated by D_0 if nonterminal D_1 generates a final letter b, or if a pair $b\,d$ already occurs in the Dyck substrings generated by D_1 or D_2. Similarly for rule 1, except that in this case an immediate pair $b\,d$ may not show up, as the rule does not contain letter d.

To check if nonterminal D_0 generates a final letter b: in the rule 1, this happens if nonterminal D_2 does so or if it is nullable; while in the rule 2 this happens only if nonterminal D_2 does so. The other semantic functions are obvious.

(c) Here is the solution, with inheritance, where more attributes (one of which is inherited) have to be added to those of grammar G_1. Here they are:

attributes to be added, if any (complete the specification)

type	*name*	*domain*	*(non)term.*	*meaning*
right	*depth*	integer ≥ 0	D	standard nesting depth (downward computation)
left	*up*	integer ≥ 0	S, D	nesting depth of pair $a\,c$ (if any) at the tree root (upward propagation)

And here is grammar G_1 extended; one can skip the semantic functions of G_1 that detect pair $a\,c$, which are here included unchanged from question (a):

#	*syntax*	*semantics* - question (c)
0:	$S_0 \rightarrow D_1$	$depth_1 = 0$ $ac_0 = ac_1$ $up_0 = up_1$

1:	$D_0 \rightarrow a\ D_1\ b\ D_2$	$depth_1 = depth_0 + 1$
		$depth_2 = depth_0$
		$c_0 = false$
		$ac_0 = c_1\ \vee\ ac_1\ \vee\ ac_2$
		if $c_1 == true$ **then**
		  $up_0 = depth_1$
		else if $ac_1 == true$ **then**
		  $up_0 = up_1$
		else if $ac_2 == true$ **then**
		  $up_0 = up_2$
		else
		  $up_0 = 0$
		end if

2:	$D_0 \rightarrow c\ D_1\ d\ D_2$	$depth_1 = depth_0 + 1$
		$depth_2 = depth_0$
		$c_0 = true$
		$ac_0 = ac_1\ \vee\ ac_2$
		if $ac_1 == true$ **then**
		  $up_0 = up_1$
		else if $ac_2 == true$ **then**
		  $up_0 = up_2$
		else
		  $up_0 = 0$
		end if

3:	$D_0 \rightarrow \varepsilon$	$c_0 = false$ $ac_0 = false$ $up_0 = 0$

The so-extended attribute grammar G_1 works as follows. The nesting depth of each parenthesis structure $a\,b$ or $c\,d$ is computed moving downwards in the usual way by means of the right attribute *depth*, with the outermost parentheses at depth 0 as suggested by the given samples. The adjacent letter pairs $a\,c$ are detected moving upwards by means of the left attributes c and ac, as it is done in the question (a). The nesting depth of the detected leftmost letter pair $a\,c$ is transported upwards by means of the left attribute *up*: this attribute is first initialized with 0 at the innermost parenthesis structures, then it is (re)assigned the depth of letter c whenever a new letter pair $a\,c$ is found leftmost, else it is simply maintained the depth of the already detected inner leftmost pair, if any. Clearly the nesting depth of the leftmost pair $a\,c$ is always given preference to be assigned to attribute *up*, see the semantic functions of the rules 1 and 2.

The so-extended attribute grammar G_1 is of type one-sweep. The reason is that the only right attribute is *depth*, which can be computed independently of all the others, and that the remaining attributes c, ac and *up* are all left and depend only on left ones, except attribute *up*, which depends on the left attributes c, ac and *up* (itself) of the child nodes, but (in the rule 1) also on the right attribute *depth* of a child node (this dependence is crucial as it transfers the inherited information flow to the synthesized one); the last however is a dependence type that is admitted for the left attributes of a one-sweep grammar.