Model solutions (2012 exam): Section A

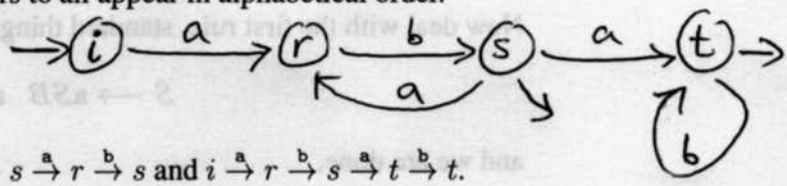
1. a. Arguably the best answer is $\mathtt{a}(\{\mathtt{a},\mathtt{b},\mathtt{c}\})^*\mathtt{a} \cup \mathtt{a}$.

You would get full credit for $\mathtt{a}(\{\mathtt{a},\mathtt{b},\mathtt{c}\})^*\mathtt{a}$ (where the one-letter word a is not deemed to be included.

For the second one, letting A be the alphabet: $(A^*\mathtt{a}A^*\mathtt{b}A^*\mathtt{c}A^*) \cup (A^*\mathtt{a}A^*\mathtt{c}A^*\mathtt{b}A^*) \cup (A^*\mathtt{b}A^*\mathtt{a}A^*\mathtt{c}A^*) \cup (A^*\mathtt{b}A^*\mathtt{c}A^*\mathtt{a}A^*) \cup (A^*\mathtt{c}A^*\mathtt{a}A^*\mathtt{b}A^*) \cup (A^*\mathtt{c}A^*\mathtt{b}A^*\mathtt{a}A^*)$

Really, you need all 6 subexpressions. You get partial credit for just using the first one, but on its own it requires the 3 letters to all appear in alphabetical order.

b. i. The diagram is straightforward.



$\mathtt{ab}(\mathtt{ab})^*(\{\epsilon\} \cup \mathtt{ab}^*)$

ii. abab has accepting paths $i \xrightarrow{\mathtt{a}} r \xrightarrow{\mathtt{b}} s \xrightarrow{\mathtt{a}} r \xrightarrow{\mathtt{b}} s$ and $i \xrightarrow{\mathtt{a}} r \xrightarrow{\mathtt{b}} s \xrightarrow{\mathtt{a}} t \xrightarrow{\mathtt{b}} t$.

2. a.

$$S \longrightarrow \mathtt{a}S'\mathtt{a} \mid S' \mid \mathtt{b}S''\mathtt{b} \mid S''$$

$$S' \longrightarrow \mathtt{a}S'\mathtt{a} \mid B$$

$$S'' \longrightarrow \mathtt{b}S''\mathtt{b} \mid A$$

$$A \longrightarrow \mathtt{a}A \mid \epsilon$$

$$B \longrightarrow \mathtt{b}B \mid \epsilon$$

It would also be acceptable to give a grammar that does not derive the empty word.

b.

$$S \Longrightarrow \mathtt{b}S''\mathtt{b} \Longrightarrow \mathtt{bb}S''\mathtt{bb} \Longrightarrow \mathtt{bb}A\mathtt{bb} \Longrightarrow \mathtt{bba}A\mathtt{bb} \Longrightarrow \mathtt{bbabb}$$

$$S \Longrightarrow S'' \Longrightarrow A \Longrightarrow \mathtt{a}A\mathtt{a} \Longrightarrow \mathtt{aa}A\mathtt{aa} \Longrightarrow \mathtt{aaaa}$$

You could also add rules like $S \to \mathtt{bbabb}$, which would make it easy to write down a derivation, and would also give a convenient explanation that the grammar is ambiguous.

c. Ambiguous since for example there are 2 (leftmost) derivations for the empty word,

$$S \Longrightarrow S'' \Longrightarrow A \Longrightarrow \epsilon$$

and

$$S \Longrightarrow S' \Longrightarrow B \Longrightarrow \epsilon$$

3. **a.** $S \longrightarrow \mathtt{a}S\mathtt{b}$, $A \longrightarrow \epsilon$.

Note that A represents $(\mathtt{a})^*$, and we can substitute into the only rule that produces an A (i.e. $S \longrightarrow \mathtt{c}A$), to get

$$S \longrightarrow \mathtt{c} \mid \mathtt{c}A' \quad \text{and} \quad A' \longrightarrow \mathtt{a} \mid \mathtt{a}A'.$$

Now deal with the first rule, standard thing is to replace it with:

$$S \longrightarrow \mathtt{a}SB \quad \text{and} \quad B \longrightarrow \mathtt{b}$$

and we are done.

b.
- variables become stack symbols
- productions become transitions, where transition says you can remove symbol corresponding to LHS variable with symbols corresponding RHS variables, at the same time as reading RHS alphabet symbol
- alphabet stays the same
- 2 states in extended PDA; initial one is accepting iff start symbol of CFG derives ϵ. Other state is entered and used after first letter of input.
- Convert extended PDA to standard PDA by using additional states.

4. **a.** It is helpful to append symbols to the start and end of the input that identify those endpoints. Then the TM could work by repeatedly scanning the input left-to-right; during a scan it will typically replace a 0 and a 1 by some other symbol X. A small number of states will keep track of whether or not a 0 has been replaced by X; whether a 1 has been replaced by X. (And if a 0 (resp. 1) has not yet been replaced, then replace the first 0 (resp. 1) that is found during the scan.) At RHS, if both a 0 and a 1 have been replaced, enter a new state that moves to LHS and repeat. If one but not the other has been replaced, if it's a 1 that has been replaced then accept else reject.

b. Not regular, but context-free. One could design a PDA that uses the stack to keep track of the number of 1's seen so far minus the number of 0's (use 2 stack symbols, say x_1 to denote each extra 1 and x_0 to denote each extra 0; the stack will always contains x_1's only or x_0's only), and has an accepting state that can be accessed if the stack contains x_1's.

Section B

Answer **two** questions in this section.

1. **a.** General guideline for getting started: you want to prove that for any language accepted by some extended DFA, there is a DFA that accepts that language. That suggests that (letting L be the language and A the extended DFA) you need to come up with a way to construct DFA A' that accepts L. And A' should somehow be constructed from A.

   The language accepted by extended DFA with start states $q_1, \ldots q_k$ is accepted by one of k DFAs obtained by selecting just one of those start states. Also, note that any word accepted by one of the k DFAs is accepted by the extended DFA. So, the language is the union of a set of regular languages, so it has a regular expression which can accordingly be converted to a standard DFA by standard method.

   An alternative approach is to let $A = (Q, A, \phi, I, T)$ (where I is a set of initial states), construct A' by adding a new initial state, $A' = (Q \cup \{i'\}, A, \phi', i', T)$, and add carefully chosen transitions to ϕ. i' should be accepting if A accepts the empty word.

   b. A DFA for the language $\{\epsilon, a\}$ needs the initial state accepting since it contains ϵ. If you want a to be accepted without any additional accepting state, you need to add a self-loop transition, which forces the DFA to accept a^*.

   c. General guideline for getting started: to prove that the two languages are the same, you want to take a word in the first language, and use the fact that it belongs to the first language to deduce that it belongs to the second. And vice versa.

   Suppose $w \in (R \cup S)^*$. Then $w = w_1 w_2 \ldots w_n$ where each w_i is in R or S (or both). Hence each w_i is in R^* or S^*, ie in $(R^* \cup S^*)$, which shows that $(R \cup S)^* \subseteq (R^* \cup S^*)^*$.
   Going the other direction, suppose $w \in (R^* \cup S^*)^*$. As before, $w = w_1 w_2 \ldots w_n$ where each w_i is in either R^* or S^*. If say $w_i \in R^*$ then $w_i = w_{i,1} \ldots w_{i,m_i}$ for $w_{i,j} \in R$. That is, each w_i can be split up into a concatenation of elements of R (or S). Hence, so can w, so $w \in (R \cup S)^*$, which proves inclusion the other way.

2. **a.** FIRST(S)={c}, FIRST(A)={a, c}, FIRST(B)={b}.
   The FOLLOW sets are all empty.

   The grammar is ambiguous; the top two rules need to be left factored. (Indeed, it is not LL(2) parsable.)

   b. For example:
   $S \longrightarrow \mathtt{c}X, X \longrightarrow \mathtt{a}, X \longrightarrow \mathtt{c}Y, Y \longrightarrow \mathtt{a}, Y \longrightarrow \mathtt{b}, Y \longrightarrow \mathtt{c}Y.$

	a	b	c
S	.	.	$S \longrightarrow \mathtt{c}X$
X	$X \longrightarrow \mathtt{a}$	.	$X \longrightarrow \mathtt{c}Y$
Y	$Y \longrightarrow \mathtt{a}$	$Y \longrightarrow \mathtt{b}$	$Y \longrightarrow \mathtt{c}Y$

   (It is helpful to figure out the following regular expression for the language: $\mathtt{c}^+\mathtt{a} \cup \mathtt{c}\mathtt{c}^+\mathtt{b}$.)

3. **a.** For two CFGs to be equivalent, they would have to be able to generate the same set of words. Now, given a single word, we can certainly check which of two CGFs generate

it (and say that they are non-equivalent if one but not the other generates the word). But there is no algorithm that is guaranteed to answer the question and terminate in finite time. (Suppose for example you try all words listed in some sequence. If you keep on finding that the two grammars agree on the word being checked, you don't know when to stop.)

b. We can easily write down a grammar G that can generate every possible word over some given alphabet. So, the statement is that it is undecidable to check whether G is equivalent to some other CFG. That's a special case of the previous problem, which is consequently undecidable.

c. Reduce from the undecidable problem of comparing unrestricted grammars, to the problem of comparing Chomsky NF grammars, which is done by just converting each grammar to Chomsky NF, then comparing those. The answer to the new problem will be the same as the answer to the problem with the unrestricted grammars, which shows that comparing Chomsky NF grammars in undecidable.

d. Recursively enumerable language L: There is an algorithm that generates all members of L (so that every word is eventually generated) and furthermore never generates non-members.

Given CFG G having (say) k rules, associate each base-k number N with a leftmost derivation obtained by, for the i-th digit d of N, apply rule d if possible (do this in ascending order of i). Every word in the language has a leftmost derivation, so if we list these base-k numbers in increasing order (first of length then of size) we obtain an associated list of members of the language given by the CFG.