Technical University of Denmark

Written examination, May 24, 2018

Course number: 02141

Aids allowed: All written aids are permitted

Exam duration: 4 hours

Weighting: 7-step scale

Answers Included

Old exam sets are not indicative of new exam sets.

Exercises on Formal Methods

Exercise 1 (25%)

Consider the program graph in Figure 1.

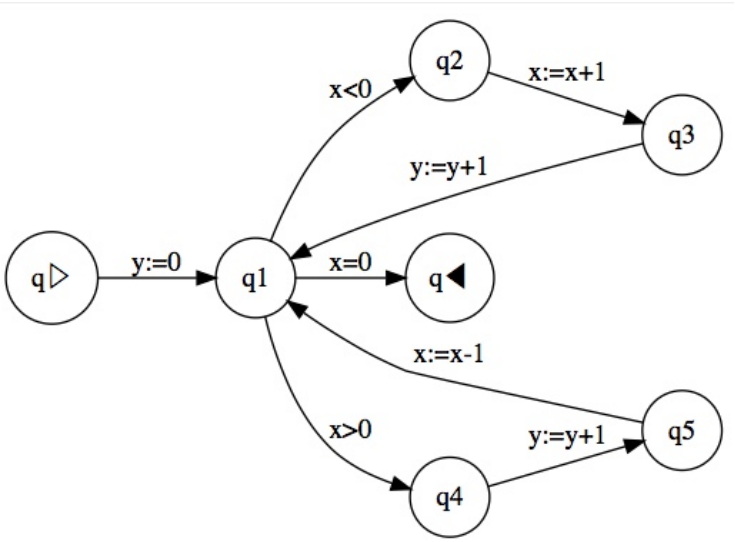


Figure 1: Program Graph for computing the absolute value.

A: Suppose that the initial memory $\sigma_{\triangleright}$ has $\sigma_{\triangleright}(\mathtt{x}) = -4$ and $\sigma_{\triangleright}(\mathtt{y}) = 27$ and imagine a complete execution sequence

$$\langle q_{\triangleright}; \sigma_{\triangleright} \rangle \stackrel{k}{\Longrightarrow} \langle q_{\blacktriangleleft}; \sigma_{\blacktriangleleft} \rangle$$

A1: What is the final memory $\sigma_{\blacktriangleleft}$?
A2: How many steps k were needed?
A3: Is the program graph deterministic?
A4: Is the program graph evolving?

Answer (4 %):
A1: $\sigma_{\blacktriangleleft}(\mathtt{x}) = 0$, $\sigma_{\blacktriangleleft}(\mathtt{y}) = 4$
A2: $k = 14$
A3: Yes, according to Proposition 1.26, because no node has a memory where more than one outgoing edge can be taken.
A4: Yes, according to Proposition 1.29, because no non-final node has a memory where none of the outgoing edges can be traversed.

B: Is there a program in Guarded Commands that gives rise to the program graph in Figure 1? If your answer is yes you should write the program in Guarded Commands. If your answer is no you should argue why this cannot be the case.

Answer (5 %): I will accept two different answers.
The first one says yes and gives the program:

```
y:=0;
do x<0 -> x:=x+1; y:=y+1
   []
   x>0 -> y:=y+1; x:=x-1
od
```

The other one says no and argues that $\mathtt{x} = 0$ does not have the form of $\neg(\mathtt{x} > 0) \wedge \neg(\mathtt{x} < 0)$ as constructed by the edges function (although it is semantically equivalent to it).

C: Recall that the absolute value $\mathsf{abs}(z)$ is defined by

$$\mathsf{abs}(z) = \begin{cases} z & \text{if } z \geq 0 \\ -z & \text{if } z < 0 \end{cases}$$

Complete the following incomplete partial predicate assignment

$$\begin{array}{rcl} \mathbf{P}(q_{\triangleright}) & = & \mathtt{x} = \underline{\mathtt{n}} \\ \mathbf{P}(q_1) & = & \cdots \\ \mathbf{P}(q_{\blacktriangleleft}) & = & \mathtt{y} = \mathsf{abs}(\underline{\mathtt{n}}) \end{array}$$

into one that is correct. Furthermore, argue that it is correct.

Answer (8 %):

$$\begin{array}{rcl} \mathbf{P}(q_{\triangleright}) & = & \mathtt{x} = \underline{\mathtt{n}} \\ \mathbf{P}(q_1) & = & \mathsf{abs}(\underline{\mathtt{n}}) = \mathtt{y} + \mathsf{abs}(\mathtt{x}) \\ \mathbf{P}(q_{\blacktriangleleft}) & = & \mathtt{y} = \mathsf{abs}(\underline{\mathtt{n}}) \end{array}$$

We need to consider 4 short path fragments:

- $q_{\triangleright}$ $\mathtt{y} := 0$ $\mathtt{q}_1$**: From** $\mathtt{x} = \underline{\mathtt{n}}$ **we get** $\mathsf{abs}(\underline{\mathtt{n}}) = \mathsf{abs}(\mathtt{x})$ **and because** $\mathtt{y} = 0$ **we also get** $\mathsf{abs}(\underline{\mathtt{n}}) = \mathtt{y} + \mathsf{abs}(\mathtt{x})$**.**
- q_1 $\mathtt{x} = 0$ $\mathtt{q}_{\blacktriangleleft}$**: From** $\mathsf{abs}(\underline{\mathtt{n}}) = \mathtt{y} + \mathsf{abs}(\mathtt{x})$ **and** $\mathtt{x} = 0$ **we get** $\mathsf{abs}(\underline{\mathtt{n}}) = \mathtt{y}$**.**
- q_1 $\mathtt{x} > 0$ $\mathtt{y} := \mathtt{y} + 1$ $\mathtt{x} := \mathtt{x} - 1$ $\mathtt{q}_1$**: The net effect is to increment** $\mathtt{y}$ **by 1 and to decrement** $\mathsf{abs}(\mathtt{x})$ **by 1 and therefore the invariant is maintained.**
- q_1 $\mathtt{x} < 0$ $\mathtt{x} := \mathtt{x} + 1$ $\mathtt{y} := \mathtt{y} + 1$ $\mathtt{q}_1$**: The net effect is to increment** $\mathtt{y}$ **by 1 and to decrement** $\mathsf{abs}(\mathtt{x})$ **by 1 and therefore the invariant is maintained.**

D: Suppose that the initial value of `x` is positive and that the initial value of `y` is positive and consider the detection of signs analysis.

D1: Construct the analysis assignment as computed by the chaotic iteration algorithm; for succinctness you may write $(+,-)$ for the abstract memory $\widehat{\sigma}$ that has $\widehat{\sigma}(\mathtt{x}) = +$ and $\widehat{\sigma}(\mathtt{y}) = -$.

Answer (6 %):

$$\begin{array}{rcl} \mathbf{A}(q_{\triangleright}) & = & \{(+,+)\} \\ \mathbf{A}(q_1) & = & \{(-,+),(0,+),(+,0),(+,+)\} \\ \mathbf{A}(q_2) & = & \{(-,+)\} \\ \mathbf{A}(q_3) & = & \{(-,+),(0,+),(+,+)\} \\ \mathbf{A}(q_4) & = & \{(+,0),(+,+)\} \\ \mathbf{A}(q_5) & = & \{(+,+)\} \\ \mathbf{A}(q_{\blacktriangleleft}) & = & (0,+)\} \end{array}$$

D2: In case $\mathbf{A}(q_2)$ and $\mathbf{A}(q_3)$ are non-empty you should explain why this is so.

Answer (2 %): The analysis does not distinguish 1 from any other positive number, so when a positive `x` is decremented by 1, there is no way to argue that it cannot become negative.

Exercise 2 (15%)

Consider the following program C in Guarded Commands:

```
i:=0;
j:=0;
do  i<n -> A[i]:=A[i]+3; i:=i+1
 [] j<m -> B[j]:=B[j]*B[j]; j:=j+1
od
```

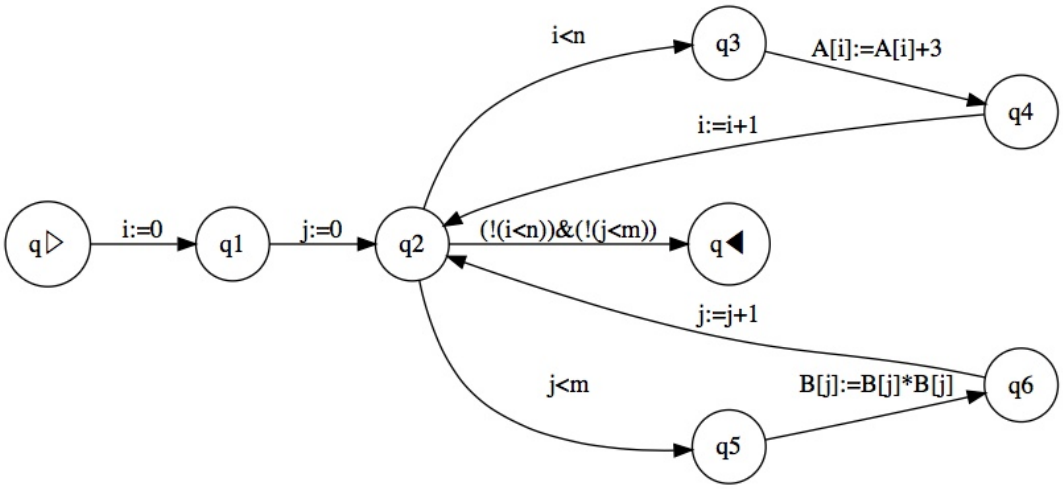
Here `A` is intended to be an array of length `n` and `B` is intended to be an array of length `m`.

A: Draw the program graph as constructed by

$$\mathbf{edges}(q_{\triangleright} \rightsquigarrow q_{\blacktriangleleft})[\![C]\!]$$

of Chapter 2.

Answer (5 %):

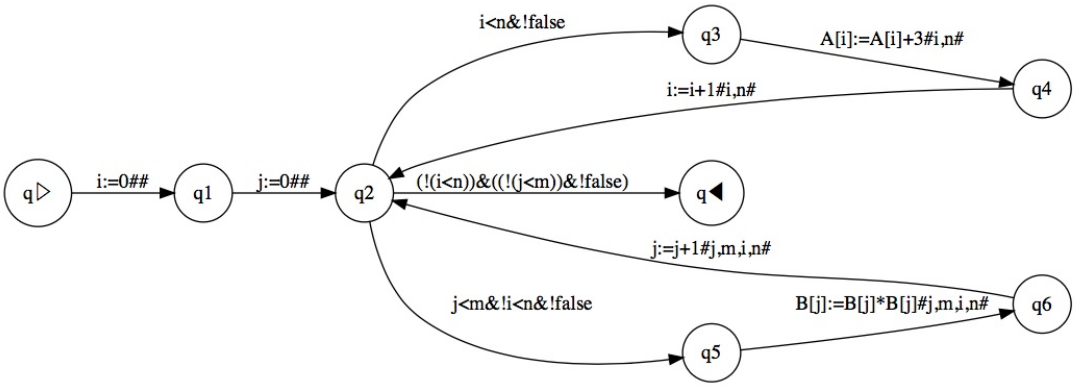


B: Draw the instrumented program graph as constructed by

$$\mathbf{edges_s}(q_{\triangleright} \rightsquigarrow q_{\blacktriangleleft})[\![C]\!](\{\,\})$$

of Chapter 5.

Answer (5 %): (For typographical reasons writing $\{\cdots\}$ as $\#\cdots\#$.)



It is essential that the program graph has been made deterministic and that the implicit flows are correctly stated.

C: Suppose that the permissible information flow is given by $\{\mathtt{i},\mathtt{A},\mathtt{n}\} \rightrightarrows \{\mathtt{i},\mathtt{A},\mathtt{n}\}$ and $\{\mathtt{j},\mathtt{B},\mathtt{m}\} \rightrightarrows \{\mathtt{j},\mathtt{B},\mathtt{m}\}$.

Is the program secure, i.e. what is the value of $\mathbf{sec}[\![C]\!](\{\,\})$?

You should motivate your answer (but do not need to exhibit the detailed computation).

Answer (5 %): The computation of sec$[\![C]\!](\{\})$ is going to check all the explicit and implicit flows made visible on the edges in the instrumented program graph. Looking at the edge with the action `j:=j+1`$\{$`j,m,i,n`$\}$ it is clear that there is an implicit flow `i`$\rightarrow$`j` that is not permitted.

Exercises on Context-free Languages

Exercise 3 (30%)

The exercises in this part of the exam are based on PingPong, a programming language whose syntax is given by the following context-free grammar. The productions are

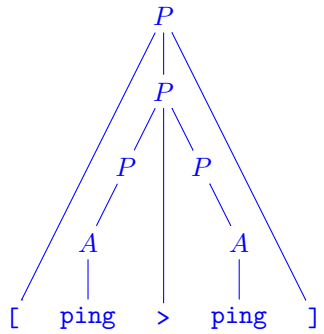
P	$\rightarrow$	A	(actions)
	$\mid$	P `>` P	(sequential composition of programs)
	$\mid$	P `+` P	(non-deterministic choice between programs)
	$\mid$	P `*` P	(parallel composition of two programs)
	$\mid$	`[`P`]`	(scoped program)
A	$\rightarrow$	`ping`	(do ping)
	$\mid$	`pong`	(do pong)

the set of non-terminal symbols is $N = \{P, A\}$, the set of terminal symbols is $T = \{\texttt{[}, \texttt{]}, \texttt{>}, \texttt{+}, \texttt{*}, \texttt{ping}, \texttt{pong}\}$, and the initial symbol is P.

(a) Show that the following program is accepted by the grammar of PingPong by providing a parse tree for it:

```
[ ping > pong ]
```

Solution: the (unique) parse tree is:



(b) Show that the grammar is ambiguous by providing one PingPong program and two distinct parse trees for it.

Solution: the key observation is that the three binary operators to combine programs give raise to ambiguities due to associativity and precedence. A simple example is

```
ping > pong + ping
```

Two distinct parse trees can be given, each encoding different associativity or precedence order between sequential composition and non-deterministic choice. The parse trees would arise from the following different groupings (denoted with parentheses here):

`(ping > pong) + ping` `ping > (pong + ping)`

(c) Consider the following associativity and precedence rules:

- `>` has the highest priority and associates to the right;
- `+` has priority over `*` and associates to the right;
- `*` has the lowest priority and associates to the left.

which of the two parse trees you provided in (b) adheres to these rules?

Solution: It depends on the answer to (b). For example, for the solution example we provided for (b), only the parse tree on the left adheres to the rules.

(d) Consider again the rules of (c). As explained in class, many grammar languages used by parser generators allow you to specify those rules in the grammar specification. Most likely, this is what you did in the mandatory assignment. Describe very briefly the grammar annotations that would implement the rules of (c) in one of the two grammar languages we saw in class (FsLexYacc or ANTLR4). Please specify which grammar language you have chosen. You don't need to give the entire grammar specification but just the part where the rules are specified.

Solution:

In FsLexYacc associativity can be specified with "`%left op`" statements for each of the operator tokens `op`, and precedence can be specified by the order of such statements (from low to high):

```
%left '*'
%right '+'
%right '>'
```

In ANTLR4 associativity can be specified with `<assoc=left/right>` annotations in the productions, while precedence can be specified by the order of the productions:

```
P : <assoc=right> P '>' P
  | <assoc=right> P '+' P
  | <assoc=left>  P '*' P
```

(e) Consider again the rules of (c). Transform the grammar of PingPong to encode the rules directly in the grammar. You can apply the techniques seen in class, i.e. enforce associativity by forbidding recursion on one of the sides, and enforce precedence by stratifying the grammar into precedence layers.

Solution: A possible solution is:

$$\begin{array}{lcl} P & \to & P \texttt{ * } Q \mid Q \\ Q & \to & R \texttt{ + } Q \mid R \\ R & \to & A \texttt{ > } R \mid \texttt{[}P\texttt{]} \texttt{ > } R \mid \texttt{[}P\texttt{]} \mid A \\ A & \to & \texttt{ping} \mid \texttt{pong} \end{array}$$

(f) Construct a Pushdown Automaton (PDA) that accepts the same language as the the grammar of PingPong described at the beginning of the exercise. Use the construction that we saw in class to translate a grammar into an equivalent PDA.

Solution: The PDA obtained accepts by empty stack and is defined by the tuple $(\{q\}, T, T \cup N, \delta, P, \{q\})$ where the transition function δ is defined as follows:

$$\begin{array}{rcll} \delta(q, P) & = & \{(q, A), (q, P\texttt{>}P), (q, P\texttt{+}P), (q, P\texttt{*}P), (q, \texttt{[}P\texttt{]})\} & \\ \delta(q, A) & = & \{(q, \texttt{ping}), (q, \texttt{pong})\} & \\ \delta(q, t, t) & = & \{(q, \epsilon)\} & \text{if } t \in T. \end{array}$$

A graphical representation of the PDA using the notation seen in class is also a valid answer.

(g) Use the PDA you have constructed to show that the example of (a) is in the language of the PDA. Show the sequence of configurations (instantaneous descriptions) of the PDA starting with the initial one $(q, \texttt{[ping > pong]}, P)$, where q is the initial state of your PDA, `[ ping > pong ]` is the input and P is the initial stack.

Solution:

$$\begin{array}{rl} & (\text{q, [ping > pong], P}) \\ \vdash & (\text{q, [ping > pong], [P]}) \\ \vdash & (\text{q, ping > pong], P]}) \\ \vdash & (\text{q, ping > pong], P > P]}) \\ \vdash & (\text{q, ping > pong], A > P]}) \\ \vdash & (\text{q, ping > pong], ping > P]}) \\ \vdash & (\text{q, > pong], > P]}) \\ \vdash & (\text{q, pong], P]}) \\ \vdash & (\text{q, pong], A]}) \\ \vdash & (\text{q, pong], pong]}) \\ \vdash & (\text{q,],]}) \\ \vdash & (\text{q, , }) \end{array}$$

(h) Design a set of datatypes that are suitable to store abstract representations (ASTs) of PingPong programs. You can take inspiration from your solution to the mandatory assignment, i.e. you can use F# types or Java classes. Show how the example program

```
[ ping > pong ] * [ ping + pong ]
```

would be represented as a value of your datatype.

Solution: In F# we could define the following types:

```
type P =
  | a of A
  | seq of (P * P)
  | choice of (P * P)
  | par of (P * P)
type A =
  | ping
  | pong
```

and the program would by represented by

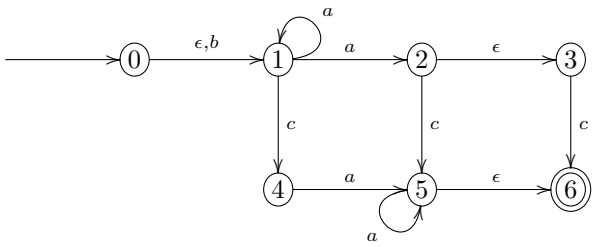
```
par(seq(a(ping),a(pong)),choice(a(ping),a(pong)))
```

Regular languages

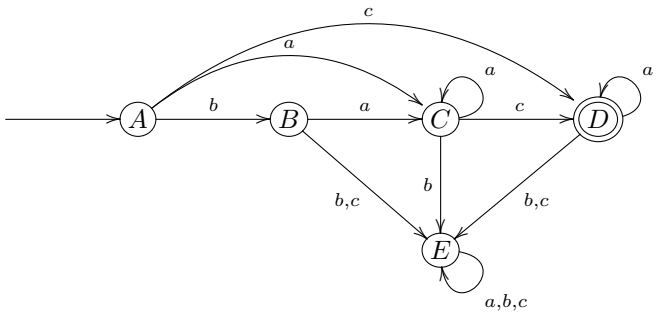
Exercise 4 (30%)

Let $\Sigma = \{a, b, c\}$ and consider the following languages:

- L_1 is given by the ϵ-NFA:



- L_2 is given by the regular expressions $(\epsilon + b)(a + c)^*$
- L_3 is given by the DFA:



- L_4 is given by $\{ba^n ca^m \mid n, m \geq 0, n \geq m\}$
- L_5 is given by the regular expression $(\epsilon + (b + \epsilon)aa^*)ca^*$

In this exercise we shall investigate the relationship between these languages.

(a) Fill out the following table with a *yes* if the string in the column is a member of the language in the row and a *no* if it is not a member of the language:

w	ϵ	c	bc	bac	bca	aac	$acaca$
L_1							
L_2							
L_3							
L_4							
L_5							

Answer (6%): The table is as follows:

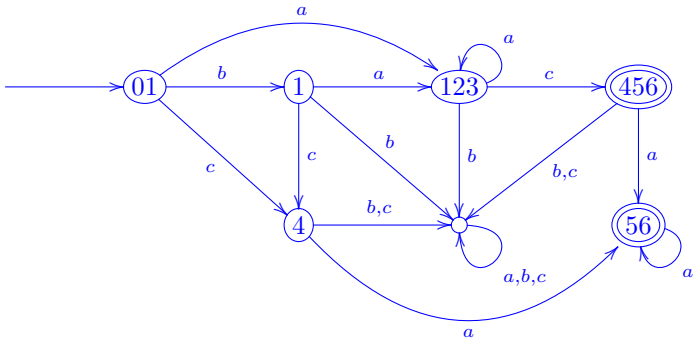
w	ϵ	c	bc	bac	bca	aac	$acaca$
L_1	*no*	*no*	*no*	*yes*	*yes*	*yes*	*no*
L_2	*yes*	*yes*	*yes*	*yes*	*yes*	*yes*	*yes*
L_3	*no*	*yes*	*no*	*yes*	*no*	*yes*	*no*
L_4	*no*	*no*	*yes*	*yes*	*no*	*no*	*no*
L_5	*no*	*yes*	*no*	*yes*	*no*	*yes*	*no*

(b) Use the subset construction algorithm (Hopcroft, Motwani and Ullman chapter 2) to construct a DFA corresponding to the ϵ-NFA for L_1.

Answer (6%): We use the subset construction algorithm of Section 2.3.5 extended to eliminate the ϵ-transitions as described in Section 2.5.5. The initial state is $ECLOSE(\{0\})$ which amounts to $\{0,1\}$. The transition relation is computed as follows:

S	a	b	c
$\{0,1\}$	$\{1,2,3\}$	$\{1\}$	$\{4\}$
$\{1,2,3\}$	$\{1,2,3\}$	$\{\ \}$	$\{4,5,6\}$
$\{1\}$	$\{1,2,3\}$	$\{\ \}$	$\{4\}$
$\{4\}$	$\{5,6\}$	$\{\ \}$	$\{\ \}$
$\{4,5,6\}$	$\{5,6\}$	$\{\ \}$	$\{\ \}$
$\{5,6\}$	$\{5,6\}$	$\{\ \}$	$\{\ \}$
$\{\ \}$	$\{\ \}$	$\{\ \}$	$\{\ \}$

The final states are those containing 6, that is, the states $\{4,5,6\}$ and $\{5,6\}$. Pictorially (not required):



(c) Prove that L_4 is not a regular language.

Answer (6%): We use the Pumping Lemma directly. So assume that L_4 is regular and consider $w = ba^N ca^N \in L_4$ where N is the constant given by the Pumping Lemma. Then there exists x, y and z such that $w = xyz$, $|xy| \leq N$, $y \neq \epsilon$ and for all $k \geq 0$ also

$xy^kz \in L_4$. **Clearly it must be the case that the c of w is in the substring z so the string xz will have fewer than N occurrences of a before the c which contradicts that $xz \in L_4$,**

(d) Fill out the following table with a *yes* if the language in the leftmost column is a subset of the language in the topmost row and a *no* if this is not the case.

$\subseteq$	L_1	L_2	L_3	L_4	L_5
L_1	*yes*				
L_2		*yes*			
L_3			*yes*		
L_4				*yes*	
L_5					*yes*

If you answer *yes*, please give an argument for why, and if you answer *no*, please give a counter example.

Answer (12%):

$\subseteq$	L_1	L_2	L_3	L_4	L_5
L_1	***yes***	yes^3	no^6	no^4	no^6
L_2	no^3	***yes***	no^5	no^2	no^5
L_3	no^6	yes^5	***yes***	no^7	yes^1
L_4	no^4	yes^2	no^7	***yes***	no^7
L_5	no^6	yes^5	yes^1	no^7	***yes***

(1) $L_3 = L_5$. **We can convert the DFA of L_3 into a regular expression. Formally, this can be done using the elimination algorithm of Section 3.2.1 that gives $((ba + a)a^*c + c)a^*$ (first eliminate B, then C). Rewriting this gives us that $L_3 = L_5$.**

(2) $L_4 \subseteq L_2$ **but** $L_2 \not\subseteq L_4$. **We observe that L_4 is a subset of ba^*ca^* and that $ba^*ca^* \subseteq b(a + c)^* \subseteq L_2$ so $L_4 \subseteq L_2$. However $L_2 \not\subseteq L_4$ because $\epsilon \in L_2$ but $\epsilon \notin L_4$.**

(3) $L_1 \subseteq L_2$ **but** $L_2 \not\subseteq L_1$. **We can convert the ϵ-NFA of L_1 (or the corresponding DFA constructed in exercise 1b)into a regular expression using the elimination algorithm of Section 3.2.1 and we get $(\epsilon + b)a^*(ca + ac)a^*$ (after some rewritings) which is a subset of $(\epsilon + b)(a + c)^*$ so $L_1 \subseteq L_2$. However $L_2 \not\subseteq L_1$ because $\epsilon \in L_2$ but $\epsilon \notin L_1$.**

(4) $L_4 \not\subseteq L_1$ **but** $L_1 \not\subseteq L_4$. **We have $L_4 \not\subseteq L_1$ since $bc = ba \in L_4$ but $bc \notin L_1$. Also $L_1 \not\subseteq L_4$ since $ca \in L_1$ but $ca = a^0ca^1 \notin L_4$ (violates $0 \geq 1$)**

(5) $L_3 \subseteq L_2$ **but** $L_2 \not\subseteq L_3$**. Recall that** $L_3 = L_5 = (\epsilon + (b + \epsilon)aa^*)ca^*$ **and** $L_2 = (\epsilon + b)(a + c)^*$ **and it is easy to see that** $L_3 \subseteq L_2$**; however** $L_2 \not\subseteq L_3$ **as** $bca \in L_2$ **but** $bca \notin L_3$**. The same holds when replacing** L_3 **with** L_5**.**

(6) $L_1 \not\subseteq L_3$ **and** $L_3 \not\subseteq L_1$**. We have** $bca \in L_1$ **but** $bca \notin L_3$ **so** $L_1 \not\subseteq L_3$**; also** $L_3 \not\subseteq L_1$ **because** $c \in L_3$ **but** $c \notin L_1$**. The same holds when replacing** L_3 **with** L_5**.**

(7) $L_4 \not\subseteq L_3$ **and** $L_3 \not\subseteq L_4$**. We have** $bc \in L_4$ **but** $bc \notin L_3$ **so** $L_4 \not\subseteq L_3$**; also** $L_3 \not\subseteq L_4$ **as strings in** L_3 **do not have to start with** b **so** $c \in L_3$ **but** $c \notin L_4$**. The same holds when replacing** L_3 **with** L_5**.**