Clayton School of Information Technology
Faculty of Information Technology

Monash University

FIT2014 Theory of Computation
SAMPLE EXAM

SOLUTIONS

2nd semester, 2013

Instructions:

10 minutes reading time.

3 hours writing time.

No books, calculators or devices.

Total marks on the exam = 120.

Sample answers in blue.

Comments in purple.

Question 1 **(4 marks)**

Annie, Henrietta, Radhanath and Williamina have been shortlisted for two jobs as computers. Let A, H, R, W be propositions with the following meanings.

A:	Annie gets one of the jobs.
H:	Henrietta gets one of the jobs.
R:	Radhanath gets one of the jobs.
W:	Williamina gets one of the jobs.

Use A, H, R and W to write a proposition, in Conjunctive Normal Form, that is True precisely when exactly two of them get jobs.

$$(A \vee H \vee R) \wedge (A \vee H \vee W) \wedge (A \vee R \vee W) \wedge (H \vee R \vee W) \wedge$$
$$(\neg A \vee \neg H \vee \neg R) \wedge (\neg A \vee \neg H \vee \neg W) \wedge (\neg A \vee \neg R \vee \neg W) \wedge (\neg H \vee \neg R \vee \neg W)$$

The first four clauses together ensure that *at least* two of them get jobs. The last four clauses together ensure that *at most* two of them get jobs.

These four names belong to four famous computers:

Annie Jump Cannon (1863–1941),
Henrietta Swan Leavitt (1868–1921),
Radhanath Sikdar (1813–1870),
Williamina Fleming (1857–1911).

Look them up!

Question 2 **(3 marks)**

Suppose you have the predicates `prolog` and `elvish`, with the following meanings:

`prolog`(X):	X knows the Prolog language.
`elvish`(X):	X knows the Elvish language.

(a) Write a universal statement in predicate logic with the meaning:

"Nobody knows both Prolog and Elvish."

$$\forall X \quad \neg(\texttt{prolog}(X) \wedge \texttt{elvish}(X))$$

Alternative answer:

$$\forall X(\neg\texttt{prolog}(X) \vee \neg\texttt{elvish}(X))$$

(b) Suppose that the statement in (a) is *False*. Starting with its negation, derive an existential statement meaning that someone knows both these languages.

If you start with the first answer given to (a) above:

Negate the answer to (a):

$$\begin{aligned}\neg\forall X \;\; \neg(\texttt{prolog}(X) \wedge \texttt{elvish}(X)) &= \exists X \neg\neg(\texttt{prolog}(X) \wedge \texttt{elvish}(X)) \\ &= \exists X (\texttt{prolog}(X) \wedge \texttt{elvish}(X))\end{aligned}$$

Question 3 **(2 marks)**

Give a regular expression for the set of all real numbers, represented in binary, that are greater than 0, less than 1, and have a finite binary representation.

(Assume that such binary numbers always have a bit before the binary point (i.e., what we would normally call the "decimal point"), and at least one bit after it.)

$\mathbf{0.}(0 \cup 1)^*\mathbf{1}$

Alternative way of writing this, using common shorthand for regexps: **0.[01]*1**

The regular expression $\mathbf{0.}(0 \cup 1)^*$ doesn't quite do the job, on two counts: firstly, it does not guarantee that there is at least one bit after the point; and secondly, it allows zero, represented as 0.000. . . 0 (with some number of zeros after the point), which the question forbids. The regular expression $\mathbf{0.}(0 \cup 1)(0 \cup 1)^*$ (which may be abbreviated as **0.[01]+**) is better, as it ensures that there be something after the point, but it still allows zero, which it shouldn't.

Question 4 **(4 marks)**

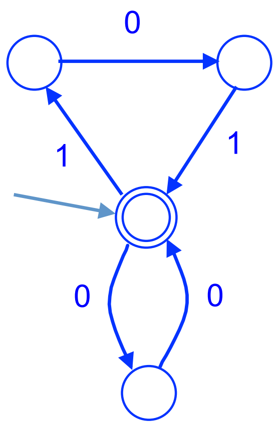
(a) Write down all strings of at most 8 letters, over alphabet {0,1}, that match the regular expression $((101)^* \cup (00))^*$.

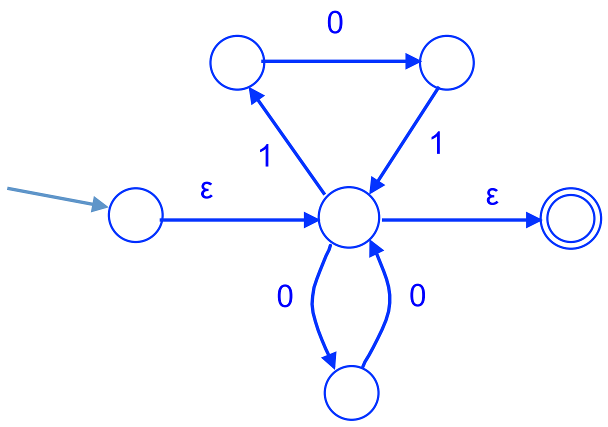
ε, 101, 101101, 00, 10100, 00101, 00101101, 10100101, 10110100, 0000, 0000101, 0010100, 1010000, 000000, 00000000.

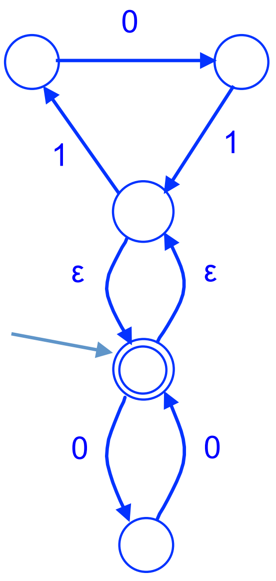
Observe that the Kleene star inside the parentheses is not needed. The given regular expression is equivalent to $((101) \cup (00))^*$.

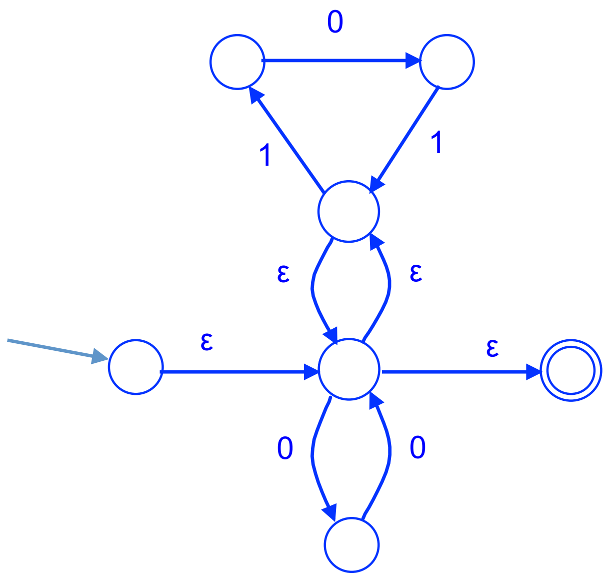
(b) Give an NFA that recognises the language described by this regular expression.

Any of the following is acceptable:









Question 5 **(3 marks)**

Prove that the class of regular languages is closed under complement.

We need to show that the complement of a regular language is also a regular language.

If L is a regular language, then by Kleene's Theorem there is a finite automaton A that recognises L. Create a new finite automaton, B, which is identical to A except that the Final states in A are not Final in B, and the non-Final states in A are Final in B. Then B will accept exactly those strings that are *not* accepted by A. So the language accepted by B is the complement of L. Therefore the complement of L, being the language accepted by some FA, is also regular, by Kleene's Theorem.

Question 6 **(3 marks)**

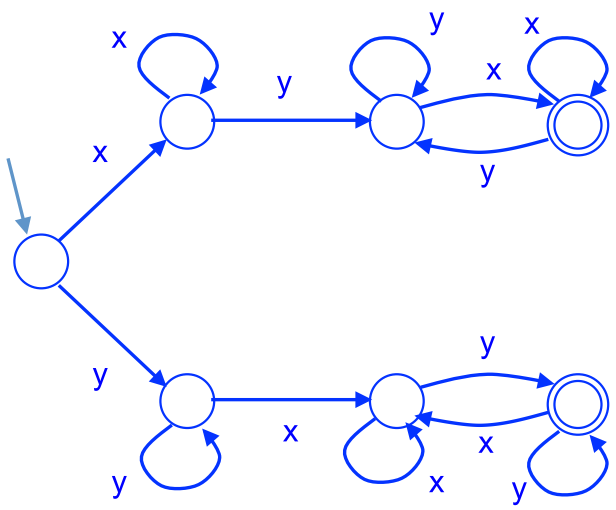
What kinds of regular languages can be described by regular expressions that do not use the Kleene star? Explain.

The Kleene star is the only regular expression operation which enables a regular expression to match an infinite number of strings. Without the Kleene star, we just have $\cup$ and concatenation, and these only allow us to match a finite number of strings.

On the other hand, any finite language can be described by some regular expression that does not use the Kleene star. Just take all the strings in the language and combine them using the alternative construct, $\cup$.

Question 7 **(3 marks)**

Let L be the language of nonempty strings over {x,y} that must start and finish with the same letter, and in the middle have at least one of the other letter. Draw a FA to recognise L.



Question 8 **(4 marks)**

Given the Finite Automaton described by the following table, find an FA with fewest states that recognises the same language.

	state	a	b
Start	1	2	6
	2	3	6
	3	6	3
	4	5	4
Final	5	4	6
Final	6	3	6

Write your new FA in the blank table below.

	state	a	b
Start	1	2	4
	2	3	4
	3	4	3
Final	4	3	4

Question 9 **(5 marks)**

(a) Prove that the language of strings of even length is regular.

Assume alphabet {a,b}.
The language of strings of even length can be described by the regular expression $((\mathtt{a} \cup \mathtt{b})(\mathtt{a} \cup \mathtt{b}))^*$. The expression within the outer parentheses, $(\mathtt{a} \cup \mathtt{b})(\mathtt{a} \cup \mathtt{b})$, matches any string of two letters. Applying the Kleene star gives a string that matches any concatenation of any finite number of strings of two letters — in other words, any string of even length. So the language of such strings is regular.

Alternatively, you could give a FA which you can show matches all strings of even length. Then appeal to Kleene's Theorem to conclude that the language of such strings is regular.

(b) Given the closure properties of regular languages, and the fact that the language of *strings of* ***even*** *length that are* ***not*** *palindromes* is not regular, prove that the language of palindromes is not regular.

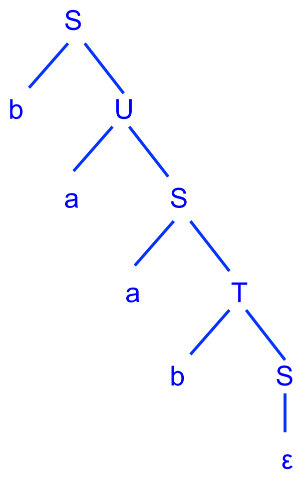
We prove this by contradiction. Assume that the language of palindromes is regular. Then its complement, the language of non-palindromes, is also regular, since the regular languages are *closed under complement*. We know from part (a) that the language of even-length strings is regular. So the intersection of the *non-palindromes* with the *even-length* strings is also regular, since these languages are both regular and the regular languages are *closed under intersection*. But this is a contradiction, as we are given that the language of even-length non-palindromes is *not* regular. Therefore our assumption, that the language of palindromes is regular, is wrong. So the language of palindromes is not regular.

Question 10 **(6 marks)**

Consider the following Regular grammar:

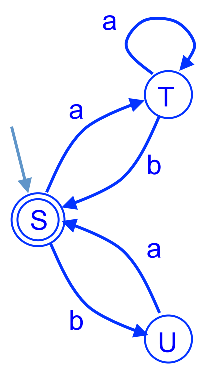
$$\begin{aligned} \mathbf{S} &\rightarrow \varepsilon \mid \mathbf{aT} \mid \mathbf{bU} && (1)\\ \mathbf{T} &\rightarrow \mathbf{aT} \mid \mathbf{bS} && (2)\\ \mathbf{U} &\rightarrow \mathbf{aS} && (3) \end{aligned}$$

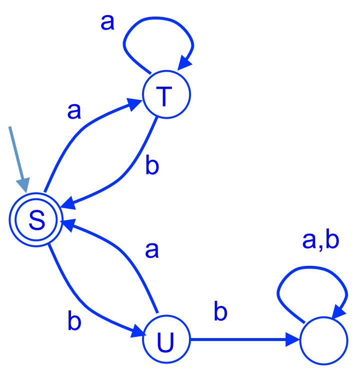
(a) Give (i) a derivation, and (ii) a parse tree, for the string **baab**



$$\begin{aligned} \mathtt{S} &\Rightarrow \mathtt{bU} && \text{by (1c)}\\ &\Rightarrow \mathtt{baS} && \text{by (3)}\\ &\Rightarrow \mathtt{baaT} && \text{by (1b)}\\ &\Rightarrow \mathtt{baabS} && \text{by (2b)}\\ &\Rightarrow \mathtt{baab} && \text{by (1a)} \end{aligned}$$

(b) Find the Finite Automaton for the language defined by the above grammar.





The FA on the left is obtained by direct conversion of the regular grammar to a FA. Strictly speaking, it is not deterministic since state U does not have a transition for the letter **b**. This is easily fixed: see the FA on the right. (In a sense, omitting transitions is a less "serious" form of nondeterminism than empty transitions or multiple transitions for the same letter, since the former does not lead to ambiguity as to how to proceed, provided a missing transition is taken to give rejection if it is attempted. That's not how we've defined rejection by FAs in this unit, though.)

(c) Give a regular expression for the language defined by the above grammar.

$$((\mathtt{aa}^*\mathtt{b}) \cup (\mathtt{ba}))^*$$

Question 11 **(3 marks)**

A string over the alphabet $\{0,+,-\}$ is said to be *balanced* if it satisfies both the following:

- (i) for any i, the first i characters in the string contain at least as many $+$ as $-$;
- (ii) the whole string has the same number of $+$ as $-$.

Give a Context-Free Grammar for the language.

$$\begin{aligned} S &\rightarrow 0 \\ S &\rightarrow SS \\ S &\rightarrow +S- \\ S &\rightarrow \varepsilon \end{aligned}$$

This is a variation on the Dyck language, with left parenthesis replaced by $+$, right parenthesis replaced by $-$, and 0s allowed to be inserted anywhere.

Question 12 **(9 marks)**

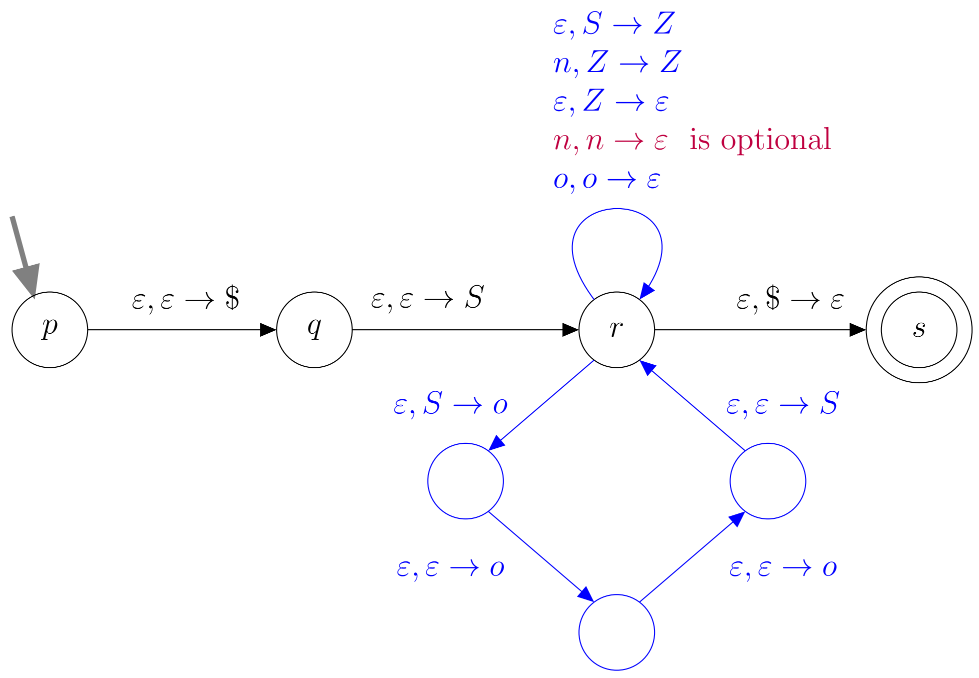
The language **Luke** has the following Context-Free Grammar:

$$\begin{aligned} S &\rightarrow Z && (1) \\ S &\rightarrow Sooo && (2) \\ Z &\rightarrow nZ && (3) \\ Z &\rightarrow \varepsilon && (4) \end{aligned}$$

(a) Give a derivation of the string $nnooooooooo$, indicating which production rule is used at each step.

$$\begin{aligned} S &\Rightarrow Sooo && \text{by (2)} \\ &\Rightarrow Soooooo && \text{by (2)} \\ &\Rightarrow Sooooooooo && \text{by (2)} \\ &\Rightarrow Zooooooooo && \text{by (1)} \\ &\Rightarrow nZooooooooo && \text{by (3)} \\ &\Rightarrow nnZooooooooo && \text{by (3)} \\ &\Rightarrow nnooooooooo && \text{by (4).} \end{aligned}$$

(b) Complete the following diagram to give a Pushdown Automaton for Luke.


$\varepsilon, S \to Z$
$n, Z \to Z$
$\varepsilon, Z \to \varepsilon$
$n, n \to \varepsilon$ is optional
$o, o \to \varepsilon$
p
$\varepsilon, \varepsilon \to \$$
q
$\varepsilon, \varepsilon \to S$
r
$\varepsilon, \$ \to \varepsilon$
s
$\varepsilon, S \to o$
$\varepsilon, \varepsilon \to S$
$\varepsilon, \varepsilon \to o$
$\varepsilon, \varepsilon \to o$

(c) Is the above CFG a regular grammar? (Explain.)

No, because there is a production rule, namely $S \to Sooo$, whose right-hand side does not consist of terminals followed by non-terminals.

(d) Is Luke a regular language? (Explain.)

Yes, because it is described by a regular expression: $\texttt{n}^*(\texttt{ooo})^*$

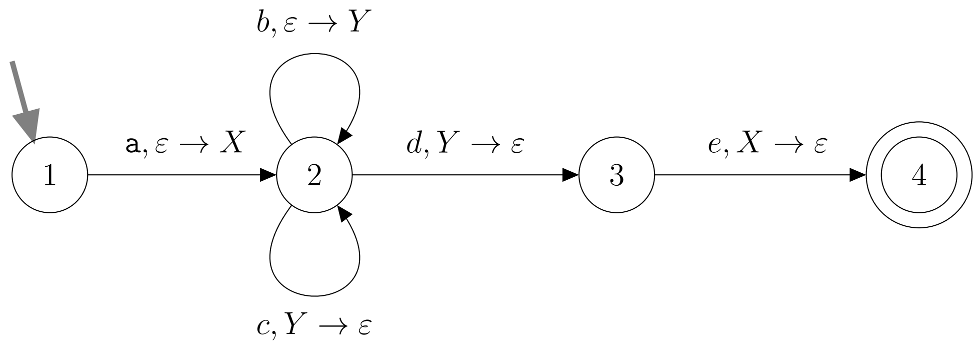
Alternatively, you could give a FA, or a regular grammar, for the language.

(e) Convert the grammar into Chomsky Normal Form.

$$\begin{array}{rcl} S & \to & \varepsilon \\ S & \to & NS \\ S & \to & SO \\ N & \to & NN \\ N & \to & n \end{array} \qquad \begin{array}{rcl} O & \to & OO \\ O & \to & O_1O' \\ O' & \to & O_2O_3 \\ O_1 & \to & o \\ O_2 & \to & o \\ O_3 & \to & o \end{array}$$

Question 13 **(5 marks)**

Find a Context-Free Grammar for the language accepted by the following PDA.



Your CFG must use only the nonterminal symbols $S, A_{11}, A_{22}, A_{33}, A_{44}, A_{14}, A_{23}$.

Write the CFG in the space below. Five production rules have already been written in, to get you started.

$S \to A_{14}$
$A_{11} \to \varepsilon$
$A_{22} \to \varepsilon$
$A_{33} \to \varepsilon$
$A_{44} \to \varepsilon$

$$\begin{aligned} A_{14} &\to aA_{23}e \\ A_{23} &\to A_{22}A_{23} \\ A_{23} &\to bA_{22}d \\ A_{22} &\to A_{22}A_{22} \\ A_{22} &\to bA_{22}c \end{aligned}$$

In this case, the symbols A_{11}, A_{33}, A_{44}, and their associated production rules, are not used.

Question 14 **(8 marks)**

(a) Prove that the language of strings representing powers of 2, in **binary** form, is regular.

This language is described by the regular expression 10^*, therefore is regular.

(b) Prove that the language of strings representing powers of 2, in **unary** form, is **not** context-free.

We prove this by contradiction, using the Pumping Lemma for Context-Free Languages.

Assume that this language is context-free, with some context-free grammar. Let k be the number of non-terminal symbols in the grammar. Let w be any word in the language of length $> 2^{k-1}$. Then, by the Pumping Lemma, w can be divided up into strings u, v, x, y, z such that v, y are not both empty, $|vxy| \leq 2^k$, and uv^nxy^nz is in the language for every n.

Observe that w is a string of 2^i ones, for some i (since w represents a power of 2, in unary). Suppose the combined length of v and y is j. Note that $j \neq 0$, since v and y are not both empty. Now, uv^2xy^2z consists of $|w| + j$ ones, since it is just w with an extra j ones (for the extra pair v, y). So its length is $2^i + j$ (since $|w| = 2^i$). If $2^i > j$, then $2^i + j$ cannot be a power of 2, so uv^2xy^2z is not in the language, which is a contradiction with the conclusion of the Pumping Lemma. (Observe that we can ensure that $2^i > j$ by just choosing w to be long enough, since the combined length j of v and y is bounded independently of the length of w (since $|vxy| \leq 2^k$). Also, in deducing that $2^i + j$ cannot be a power of 2, we are using the fact that $j \neq 0$.)

So the assumption that the language is context-free is incorrect. Therefore it is not context-free.

Question 15 **(3 marks)**

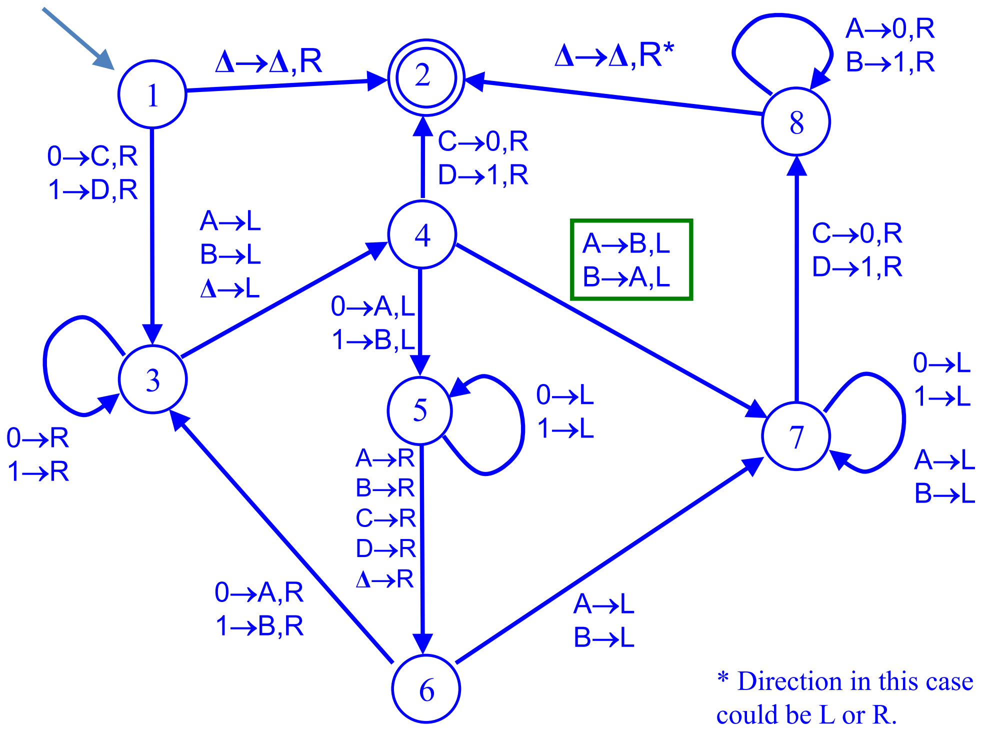
State two important results that can be proved using the Chomsky Normal Form for Context-Free Grammars.

The Cocke-Yonger-Kasami (CYK) Algorithm, which enables us to parse strings for any context-free language, uses Chomsky Normal Form.

The Pumping Lemma for CFLs (see above) also uses Chomsky Normal Form.

Question 16 **(6 marks)**
Write a Turing machine that **flips the middle bit** (i.e., changes 0 to 1, and 1 to 0) of a binary string of odd length, and leaves a string of even length unchanged.

For example, if the input string is 0111110, then the output must be 0110110. If the input string is 011110, then the output is also 011110.



Outline of how this works:

In the above Turing machine: you start by replacing the first and last bits by letters, with 0 being replaced by A or C, and 1 being replaced by B or D. The positions of these bits are easy to recognise. The first bit is where you start ($1 \to 2$), and you find the last bit by moving to the right (loop at State 3) until you reach a blank, then you go one step to the left ($3 \to 4$) so you know you're at the last bit, which you then change to a letter ($4 \to 5$). Now that the first and last bits are letters, you go all the way back (loop at State 5) until you reach another letter, then go one step back from it ($5 \to 6$). You are now at the leftmost bit; there was a bit further to the left but it has been replaced by a letter.

You keep going like this, shuttling back and forth along the tape, replacing each pair of outermost bits by letters, and passing over the bits between them, until all bits have been replaced by letters. We find ourselves repeatedly going around the circuit 6,3,4,5,6. The bits in the middle, between the letters, decrease in number. If the original bit-string is of even length, then we leave the circuit at 6. If it is of odd length, we leave it at 4. Then, using the loop at State 7, we move back (i.e., leftwards) to the start, and we can recognise when we're at the start because in this position the bit was replaced by C or D, rather than A or

B. Then (State 8) we go all the way along the tape (rightwards), replacing every A or C by 0, and every B or D by 1. When we reach a blank, we stop.

In this TM, the flip of the middle bit (if such exists) is done in transition $4 \to 7$ (see green box), using the letter that (by then) represents that bit.

Strings consisting of just a single bit are a special case. They are the only input strings that use the transition $4 \to 2$. They cause the Turing machine to do the transitions $1 \to 3 \to 4 \to 2$, which together have the effect of flipping the bit.[1]

Question 17 **(1 marks)**

The *characteristic function* f_L of a language L over some alphabet is defined by:

$$f_L(w) = \begin{cases} 1, & x \in L, \\ 0, & x \notin L, \end{cases}$$

for any string w over the alphabet.

State the property that f_L must have, for the language L to be decidable.

f_L must be *computable*.

Question 18 **(4 marks)**

For each of the following decision problems, indicate whether or not it is decidable.

You may assume that, when Turing machines are encoded as strings, this is done using the Code-Word Language (CWL).

Decision Problem	your answer (tick **one** box in each row)	
Input: Turing machines M and N. Question: Are the encoded forms of M and N identical?	✓ Decidable	Undecidable
Input: Turing machines M and N. Question: Do M and N have the same time complexity?	Decidable	✓ Undecidable
Input: a Turing machine M. Question: Does M correctly determine whether or not its input string is a palindrome?	Decidable	✓ Undecidable
Input: a Turing machine M, and a string w. Question: Does M ever change any letter of w on the tape?	Decidable	✓ Undecidable

[1] Thanks to FIT2014 tutor Rebecca Young for suggesting the modification to deal with this case.

Question 19 **(9 marks)**

The Venn diagram at the left shows several classes of languages. For each language (a)–(l) in the list below, indicate which classes it belongs to, and which it doesn't belong to, by placing its corresponding letter in the correct region of the diagram.

If a language does not belong to any of these classes, then place its letter above the top of the diagram.

(a) The Dyck language.

(b) The set of all even numbers, represented in binary.

(c) The set of all correctly formed arithmetic expressions, using integers and the symbols $+$, $-$, $\times$, $/$, and parentheses.

(d) The Code-Word Language (CWL).

(e) The set of all encodings of Turing machines (encoded using strings from CWL).

(f) DOUBLEWORD, the set of all strings consisting of a string concatenated with itself.

(g) The set of all palindromes (i.e., strings that are the same forwards or backwards).

(h) The set of all Turing machines that accept every binary string.

(i) The set of all regular expressions.

(j) The set of all polynomials (with any number of variables) with an integer root.

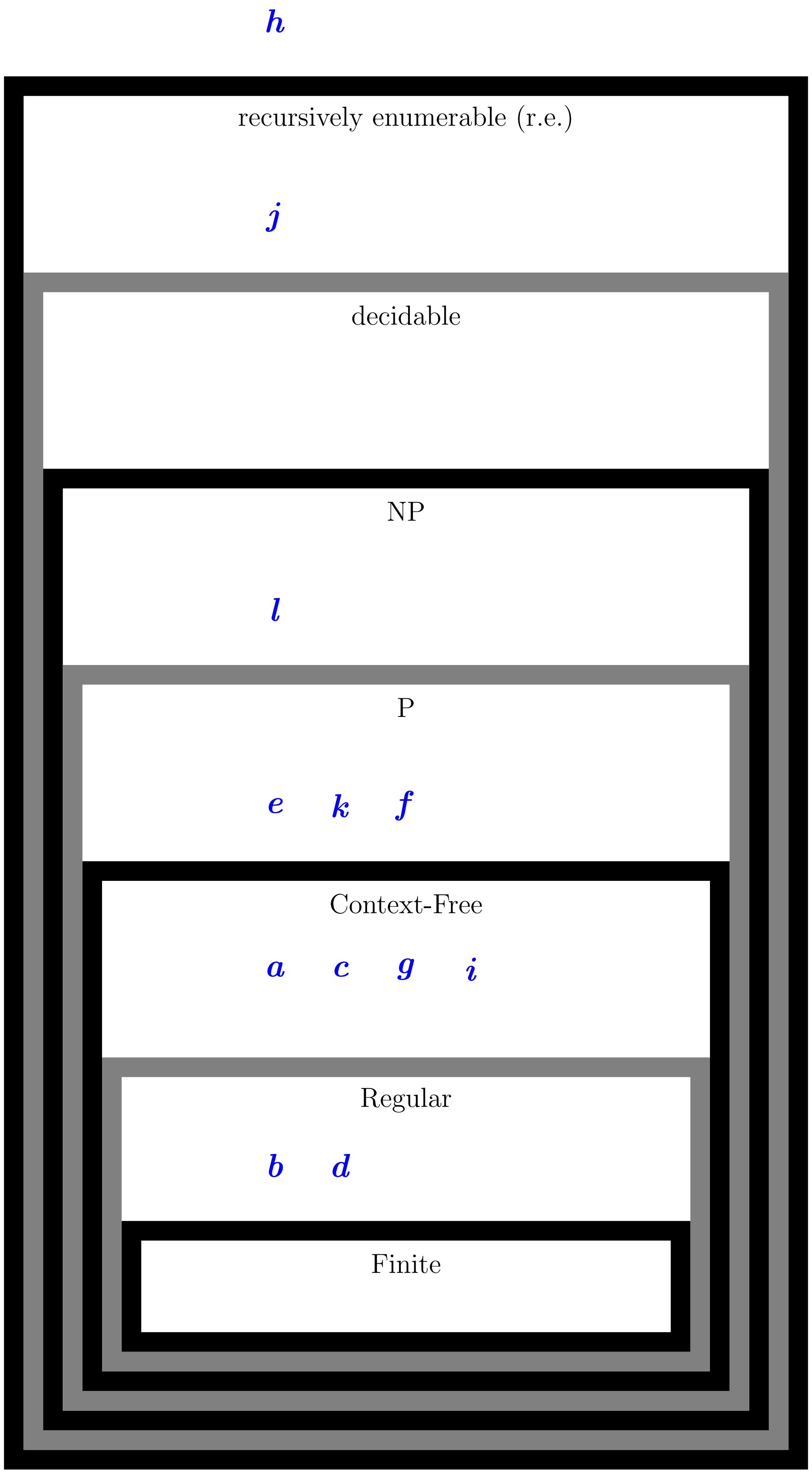
(k) The set of all satisfiable Boolean expressions in Conjunctive Normal Form with at most two literals in each clause.

(l) The set of all satisfiable Boolean expressions in Conjunctive Normal Form with at most three literals in each clause.

Which, if any, of these languages are NP-complete?

The language ***l*** is NP-complete.

This is easy to show, by polynomial-time reduction from 3SAT.


h
recursively enumerable (r.e.)
j
decidable
NP
l
P
e k f
Context-Free
a c g i
Regular
b d
Finite

Question 20 **(7 marks)**
Prove that the following problem is undecidable.

Input: a Turing machine M, and a positive integer t.
Question: Is there an input string x of length at least t such that, if M is run on x, it eventually halts?

You may use the fact that the halting problem is undecidable.

NEW SOLUTION (with mapping reduction):
We give a mapping reduction from the Halting Problem to the given problem.
We first define it, as a function f, and then show it is a mapping reduction.

This function f takes, as input, an input for the Halting Problem, i.e., a pair (T, w) where T is a Turing machine (appropriately encoded), and w is a string which is treated as an input to T. From (T, w), the function f constructs the following program, which we call M:

M'

```
1. Input: a string $x$.
2. Simulate $T$ on input $w$. (This is hardcoded. So we aren't taking $w$
as input to $M$, but rather, we have lines of code (or Turing machine
instructions) that provide $w$ as input to this simulation of $T$.)
3. Test if $|x|$ is a multiple of 29. (The choice of 29 is arbitrary! All we
want to ensure is that there is an infinite sequence of strings $x$ that
satisfy this test.)
      If so, halt; otherwise, loop forever.
```

The function f is computable, because all the parts of f can be computed from (T, w).

- Note that M' is <u>not</u> an algorithm for computing the function f. Instead, M' is, itself, the *output* of f, when the input to f is (T, w).

Observe that:

- If T eventually halts for input w, then the simulation in Step 2 of M will eventually stop. Then M goes on to Step 3. For any t, there will be some multiple of 29 that is greater than t (in fact, infinitely many, but we only need one). So, there is some x which, if given to M, causes it to halt eventually.

- On the other hand, if T loops forever for input w, then M will be forever stuck in Step 2. Therefore, there is no input x which causes M to eventually halt, and therefore certainly no such input with length $\geq t$.

So, T halts for input w if and only if there is some input x with $|x| \geq t$ such that M halts for input x. In other words, (T, w) is a YES-input for the Halting Problem if and only if it is a YES-input for the given problem.
Therefore f is a mapping reduction.

Since we have a mapping reduction from the Halting Problem, which is known to be undecidable, to the given problem, it follows that the given problem is also undecidable.

OLD SOLUTION (without mapping reduction):
Let (T, w) be an input to the Halting Problem. So T is a Turing machine (appropriately encoded), and w is a string which can be treated as an input to T. For the Halting Problem, we want to know whether or not T eventually halts, when its input is w.
We construct from (T, w) a program M which runs as follows:

1. Input: a string x.
2. Simulate T on input w. (This is hardcoded. So we aren't taking w as input to M, but rather, we have lines of code (or Turing machine instructions) that provide w as input to this simulation of T.)
3. Test if $|x|$ is a multiple of 29. (The choice of 29 is arbitrary! All we want to ensure is that there is an infinite sequence of strings x that satisfy this test.)
    If so, halt; otherwise, loop forever.

Observe that:
If T eventually halts, on input w, then the simulation in Step 2 of M will eventually stop. Then M goes on to Step 3. For any t, there will be some multiple of 29 that is greater than t (in fact, infinitely many, but we only need one). So, there is some x which, if given to M, causes it to halt eventually.
On the other hand, if T loops forever for input w, then M will be forever stuck in Step 2. Therefore, there is no input x which causes M to eventually halt, and therefore certainly no such input with length $\geq t$.

So, T halts for input w if and only if there is some input x with $|x| \geq t$ such that M halts for input x.

This tells us that, if the problem stated in the question is decidable, then we could use a decider for it to construct a decider for the Halting Problem. To see this, suppose D is a decider for the problem stated in the question. Suppose we are given (T, w), as input to the Halting Problem. From (T, x), construct M as described above. This construction is computable. Then use D to determine whether or not M has an input x, with $|x| \geq t$, for which it eventually halts. The answer D gives us becomes our answer to the Halting Problem regarding (T, x). So we are done.

Therefore the problem stated in the question is undecidable.

Question 21 **(5 marks)**

For this question, recall that the composition of two polynomial-time reductions is again a polynomial-time reduction, and that the notation $\leq_p$ indicates the existence of a polynomial-time reduction.

Prove by induction on n that, if $L_1, \ldots, L_n$ are languages, and $L_i \leq_p L_{i+1}$ for all i in the range $1 \leq i \leq n-1$, then $L_1 \leq_p L_n$.

Inductive hypothesis:

There is a polynomial-time reduction from L_1 to L_i.

Base case:

If $i = 1$, then the identity map (i.e., the polynomial-time reduction that "does nothing", just mapping everything to itself), establishes that $L_1 \leq L_1$.
This is the *reflexive* property of polynomial-time reducibility.
Alternatively, you can take $i = 2$ as the base case, since we are given that $L_1 \leq L_2$.

Inductive step:

Suppose the inductive hypothesis holds when $i = j$, where $j \geq 2$ (or $j \geq 3$, if you used $i = 2$ as your base case). We want to show that it holds for $i = j + 1$.

The inductive hypothesis tells us that there is a polynomial-time reduction from L_1 to L_j.

We can assume (from the information given in the question) that there is a polynomial-time reduction from L_j to L_{j+1}.

These two polynomial-time reductions combine (i.e., compose) to give a polynomial-time reduction from L_1 to L_{j+1}.

Conclusion:

The result therefore follows, by the Principle of Mathematical Induction.

Question 22 **(6 marks)**

(a) Define the class of *NP-complete* languages.

A language L is *NP-complete* if it belongs to NP and every language L' in NP is polynomial-time reducible to L.

(b) Prove that, if K is NP-complete, $K \leq_p L$ and $L \in \text{NP}$, then L is NP-complete.

We are given that $L \in \text{NP}$. So, to show that L is NP-complete, it remains to show that, for any language $J \in \text{NP}$, we have $J \leq_p L$.

Take any $J \in \text{NP}$. Since K is NP-complete, we know $J \leq_p K$. But we are given that $K \leq_p L$. By transitivity, $J \leq_p K$ and $K \leq_p L$ implies $J \leq_p L$. So any language in NP is polynomial-time reducible to L.

We conclude that L is NP-complete.

Question 23 **(17 marks)**

Consider the language CUBIC SUBGRAPH, which consists of all graphs G which have a subgraph, with at least one edge, whose vertices all have degree 0 or 3.

(a) Prove that the language CUBIC SUBGRAPH is in NP.

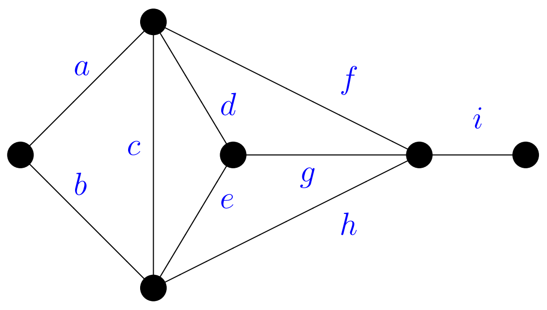
```
Verifier for CUBIC SUBGRAPH:

Input: Graph $G$

Certificate: a subgraph $Y$ of $G$ (as a set of edges).

Check that each edge given in $Y$ is indeed an edge of $G$. If $Y$ is empty, Reject.

For each vertex $v$ in $G$:
    Count the number of edges of $G$ that are incident with $v$ *and* are also in $Y$.
    If this number is 0 or 3, continue; otherwise, Reject.

If end of loop reached, then the test in each loop iteration must have been passed, so Accept.

End of Verifier.
```

A graph G belongs to CUBIC SUBGRAPH if and only if some subgraph has all vertices of degree 0 or 3, which is precisely the condition under which there exists a certificate Y which causes G to be accepted by the above verifier. So this is indeed a verifier for CUBIC SUBGRAPH.

To see that it runs in polynomial time: observe firstly that the main loop is executed at most n times (where n is the number of vertices of G). In each iteration, each edge incident with v is examined, and there are at most $n-1$ of these. For each such edge, we see if it is in the certificate Y. This means looking it up in Y, which we may think of as a list of at most m edges (where m is the number of edges of G). This takes time $O(m)$, even with just a simple list for the certificate. So total time complexity is $O(n \times (n-1) \times m) = O(n^2 m)$ which is at most a polynomial in the input size. (E.g., if input is given as an adjacency matrix, then input size is n^2, and using $m = O(n^2)$, we see that the time complexity of verification is at most quadratic in the input size.)

Smarter algorithms and data structures will give lower time complexity, but such cleverness is not necessary to show that the algorithm runs in polynomial time.

Now, let W be the following graph.



(b) Construct a Boolean expression E_W in Conjunctive Normal Form such that the satisfying truth assignments for E_W correspond to solutions to the CUBIC SUBGRAPH problem on the above graph W. (I.e., they correspond to subgraphs of W for which every vertex has degree 0 or 3 in the subgraph.)

I've marked variable names on the edges in the picture of W above.

The interpretation of each variable is that it is True if that edge is included in the cubic subgraph, and False otherwise.

The clauses are given in the following table.

What you want to say	How to say it, using clauses
For each vertex v, it can't have just **one** incident edge in Y. In other words, if one of its incident edges is in Y, then at least one other such edge must be in Y too.	$\neg a \vee b,\ \neg b \vee a,$ $\neg a \vee c \vee d \vee f,\ \neg c \vee a \vee d \vee f,\ \neg d \vee a \vee c \vee f,$ $\neg f \vee a \vee c \vee d,$ $\neg b \vee c \vee e \vee h,\ \neg c \vee b \vee e \vee h,\ \neg e \vee b \vee c \vee h,$ $\neg h \vee b \vee c \vee e,$ $\neg d \vee e \vee g,\ \neg e \vee d \vee g,\ \neg g \vee d \vee e,$ $\neg f \vee g \vee h \vee i,\ \neg g \vee f \vee h \vee i,\ \neg h \vee f \vee g \vee i,$ $\neg i \vee f \vee g \vee h,$ $\neg i.$
For each vertex v, it can't have just **two** incident edges in Y. In other words, if any two of its incident edges are in Y, then at least one other such edge must be in Y too.	$\neg a \vee \neg b,$ $\neg a \vee \neg c \vee d \vee f,\ \neg a \vee \neg d \vee c \vee f,\ \neg a \vee \neg f \vee c \vee d,$ $\neg c \vee \neg d \vee a \vee f,\ \neg c \vee \neg f \vee a \vee d,\ \neg d \vee \neg f \vee a \vee c,$ $\neg b \vee \neg c \vee e \vee h,\ \neg b \vee \neg e \vee c \vee h,\ \neg b \vee \neg h \vee c \vee e,$ $\neg c \vee \neg e \vee b \vee h,\ \neg c \vee \neg h \vee b \vee e,\ \neg e \vee \neg h \vee b \vee c,$ $\neg d \vee \neg e \vee g,\ \neg d \vee \neg g \vee e,\ \neg e \vee \neg g \vee d,$ $\neg f \vee \neg g \vee h \vee i,\ \neg f \vee \neg h \vee g \vee i,\ \neg f \vee \neg i \vee g \vee h,$ $\neg g \vee \neg h \vee f \vee i,\ \neg g \vee \neg i \vee f \vee h,\ \neg h \vee \neg i \vee f \vee g.$
For each vertex v, it can't have **four or more** incident edges in Y. In other words, for any four of its incident edges, at least one must *not* be in Y.	$\neg a \vee \neg c \vee \neg d \vee \neg f,$ $\neg b \vee \neg c \vee \neg e \vee \neg h,$ $\neg f \vee \neg g \vee \neg h \vee \neg i.$
Y has at least one edge	$a \vee b \vee c \vee d \vee e \vee f \vee g \vee h \vee i$

E_W is just the conjunction of all the clauses listed above.

(c) Give a polynomial-time reduction from CUBIC SUBGRAPH to SATISFIABILITY.

Input: Graph G.

For each edge e:
    Create a new variable x_e.

For each vertex v:

1. For each edge e incident with v, create a clause
$$\neg x_e \vee x_{f_1} \vee x_{f_2} \vee \cdots,$$
where $f_1, f_2, \ldots$ are all the other edges incident with v (apart from e).

   This clause says that "if e in Y, then some other edge at v must be in Y too".

   Together, all the clauses created in this loop say that Y cannot meet v with **exactly one** edge.

2. For each pair of edges e_1, e_2 incident with v, create a clause
$$\neg x_{e_1} \vee \neg x_{e_2} \vee x_{f_1} \vee x_{f_2} \vee \cdots,$$
where $f_1, f_2, \ldots$ are all the other edges incident with v (apart from e_1 and e_2).

   This clause says that "if e_1 and e_2 are both in Y, then some other edge at v must be in Y too".

   Together, all the clauses created in this loop say that Y cannot meet v with **exactly two** edges.

3. For each 4-tuple of edges e_1, e_2, e_3, e_4 incident with v, create a clause
$$\neg x_{e_1} \vee \neg x_{e_2} \vee \neg x_{e_3} \vee \neg x_{e_4}.$$
This clause says that "e_1, e_2, e_3 and e_4 cannot all be in Y".

   Together, all the clauses created in this loop say that Y cannot meet v with **four or more** edges.

4. Create a clause consisting of the disjunction of all our variables:
$$x_{e_1} \vee x_{e_2} \vee \cdots \vee x_{e_m},$$
where the edges of the graph are $e_1, e_2, \ldots, e_m$.

Let $\phi :=$ conjunction of all clauses created so far.

Output: ϕ.

(d) Give the usual name for the set of all languages that are polynomial-time Turing reducible to SATISFIABILITY.

This is just **NP**.

If a language L is in NP, then it must be polynomial-time reducible to SAT (or to any other NP-complete language, for that matter), by the definition of NP-completeness.

On the other hand, if a language is polynomial-time reducible to SAT, then it is polynomial-time reducible to a language in NP (since SAT is in NP), so it must be in NP itself.

So NP is precisely the set of languages polynomial-time reducible to SAT.

(e) If it were shown that all algorithms for CUBIC SUBGRAPH take exponential time, what would you conclude about the time complexity of SATISFIABILITY?

If CUBIC SUBGRAPH takes exponential time, then it is not possible for an algorithm for SAT to take polynomial time.

This is because we know that CUBIC SUBGRAPH $\leq_P$ SAT (by part (c) of this question), which implies that if SAT were in P, then CUBIC SUBGRAPH would be in P too.

From SAT $\notin$ P, it would follow that P $\neq$ NP, since we know that SAT is in NP.

Blank Page for Working

END OF EXAMINATION