Monash University
Faculty of Information Technology
2^{nd} Semester 2017

FIT2014 Theory of Computation
Solutions for Tutorial 2
Regular Languages, Inductive Definitions and Finite Automata

Although you may not need to do all the many exercises in this Tutorial Sheet, it is still important that you attempt all the main questions and a selection of the Supplementary Exercises.

Even for those Supplementary Exercises that you do not attempt seriously, you should still give some thought to how to do them before reading the solutions.

1.

(i) Inductive basis: $H_1 = 1 = \log_e e > \log_e 2$, using the fact that $e > 2$.

(ii) Our inductive hypothesis is that $H_n \geq \log_e(n+1)$ is true for n. We need to use this to show that n can be replaced by $n+1$ in this inequality, i.e., $H_{n+1} \geq \log_e((n+1)+1)$.

$$
\begin{aligned}
H_{n+1} &= 1 + \frac{1}{2} + \cdots + \frac{1}{n+1} && \text{First, we need to relate this to } H_n. \\
&= \left(1 + \frac{1}{2} + \cdots + \frac{1}{n}\right) + \frac{1}{n+1} \\
&= H_n + \frac{1}{n+1} && \text{We've now expressed } H_{n+1} \text{ in terms of } H_n. \\
& && \text{So we can apply the Inductive Hypothesis.} \\
&\geq \log_e(n+1) + \frac{1}{n+1} && \text{by the Inductive Hypothesis} \\
&\geq \log_e(n+1) + \log_e\left(1 + \frac{1}{n+1}\right) && \text{using } \log_e(1+x) \leq x, \text{ with } x = \tfrac{1}{n+1} \\
&= \log_e(n+1) + \log_e\left(\frac{n+2}{n+1}\right) \\
&= \log_e\left((n+1) \cdot \frac{n+2}{n+1}\right) \\
&= \log_e(n+2) \\
&= \log_e((n+1)+1).
\end{aligned}
$$

So, we've shown that, if the claimed inequality holds for n, then it holds for $n+1$.

(iii) By the Principle of Mathematical Induction, the claimed inequality must hold for all n.

Further properties of the harmonic numbers, and their applications in computer science, may be found in:

- Donald E. Knuth, *The Art of Computer Programming. Volume 1: Fundamental Algorithms. Third Edition.* Addison-Wesley, Reading, Ma., USA, 1997. See Section 1.2.7, pp. 75–79.

2. **abab**, **baab**, **abaaab**, **abbbab**, **baaaab**, **babbab**, **abaaaaab**, **abaabbab**, **abbbaaab**, **abbbbbab**, **baaaaaab**, **baaabbab**, **babbaaab**, **babbbbab**

3.

(i) $(\mathbf{a} \cup \mathbf{b})(\mathbf{a} \cup \mathbf{b})$ or $\mathbf{aa} \cup \mathbf{ab} \cup \mathbf{ba} \cup \mathbf{bb}$

(ii) $\mathbf{a}^*\mathbf{ba}^*\mathbf{ba}^* \cup \mathbf{a}^*\mathbf{ba}^*\mathbf{ba}^*\mathbf{ba}^*$

(iii) $(\mathbf{a} \cup \mathbf{b})^*(\mathbf{aa} \cup \mathbf{bb})$

(iv) $(\mathbf{a} \cup \mathbf{b})^*(\mathbf{ab} \cup \mathbf{ba}) \cup \mathbf{a} \cup \mathbf{b} \cup \epsilon$

(v) $(\mathbf{b} \cup \epsilon)(\mathbf{ab})^*\mathbf{aa}(\mathbf{ba})^*(\mathbf{b} \cup \epsilon) \cup (\mathbf{a} \cup \epsilon)(\mathbf{ba})^*\mathbf{bb}(\mathbf{ab})^*(\mathbf{a} \cup \epsilon)$

(vi) $(\mathbf{aa} \cup \mathbf{ab} \cup \mathbf{ba} \cup \mathbf{bb})^*$

(vii) $\mathbf{a}^*((\mathbf{b} \cup \mathbf{bb})\mathbf{aa}^*)^*(\mathbf{b} \cup \mathbf{bb} \cup \epsilon)$

4.

(a)

First, let's consider *"restricted MS Word expressions"*, where we allow all the constructs given in the definition in the document *except* for $\backslash n$ (described just before the table).[1]

Restricted MS Word expressions allow concatentation, grouping, and also the + variant of the Kleene $*$ (although note that they use non-standard notation, e.g., @ for +). They fall short of regular expressions in two ways:

- They do not allow *alternatives*, i.e., $\cup$, except for single characters. So you can have, for example, a $\cup$ b $\cup$ c. This would be denoted by the expression [abc] or [a-c]. But you cannot have ab $\cup$ ba. You can have an expression that matches just **cat** and **hat**: the expression [ch]at achieves this. But you can't write it as an alternative between the two *strings* cat and hat.

  In fact, careful consideration of the constructs reveals that there is no MS Word expression for the language {**ab**,**ba**}.

  Proof of this claim:

  Let R be a restricted MS Word expression for the language {**ab**,**ba**}.

  In this language, every string has two letters, and these two-letter strings have no repetitions. So R contains no repetition construct that allows a non-empty string to be repeated at all. The only possible use of a repetition construct (except for those that can only match empty strings) is $r\{1\}$ or $r\{0,1\}$, where r is some restricted MS Word expression (and r must match some non-empty string, at least). Now, if R contains $r\{0,1\}$ then it must match at least two strings of differing lengths, and our language has no such strings. So the only repetition construct allowed in R is $r\{1\}$. But this is not really a repetition construct at all: we can delete the "$\{1\}$" here, and just use r. So we can assume that R contains no repetition construct.

  With repetition excluded, the only other way to have two strings starting with different letters, **a** and **b**, in the language, is for R to start with a construct of the form [**ab**...]. Let R' be the rest of R, after this. So $R = [\mathbf{ab}\ldots]R'$. Let $\mathbf{a}s$ be some string, starting with **a**, that is matches R. Then the substring s matches R'. Then the string $\mathbf{b}s$ also matches R. So the language matched by R must contain two identical strings that only differ in their first letter. The language {**ab**,**ba**} contains no such two strings, and therefore cannot be matched by R, a contradiction. So there is no restricted MS Word expression for the language {**ab**,**ba**}.

[1] By the way, there appears to be an error in the first bullet point in the Notes just above the table, in the documentation referred to by this question. It implies that the 'Find whole words only' option is turned on when you check 'Use wildcards' in order to search for MS Word expressions. In fact, the 'Find whole words only' option must surely be turned *off* in this case (although 'match case' is turned on, as claimed).

- In repetition, they do not allow *zero* repetitions. All the repetition constructs — {n}, {n,}, {n,m}, @ — require one or more matches ($n > 0$). (The document does not explicitly say this, but you can check this using a recent version of Word (e.g., 2011).) The wildcard $*$ (note, in MS Word expressions, this is *not* the Kleene $*$) can match zero or more characters, but it matches arbitrary strings of any length, and cannot be used to match zero or more repetitions of a specific string.

  It follows that the language described by the regular expression a* cannot be described by any MS Word expression: to get the repetition, there has to be a repetition construct, but then there must always be at least one occurrence of the expression being repeated. This case might be rescued if general alternatives and the empty string expression ε were allowed, since we could then write $\varepsilon \cup$ a+. But neither of these are possible in MS Word expressions.

Now, what about \$n$? This actually allows MS Word expressions to describe some non-regular languages. For example, the Pumping Lemma for Regular Expressions can be used to show that $\{\mathbf{a}^n\mathbf{b}\mathbf{a}^n \mid n \geq 1\}$ is not regular, but it is described by the MS Word expression (a@)b\1 .

But the \$n$ construct does not help us recognise the simple languages we met earlier: {**ab**,**ba**} and $\{\mathbf{a}^n \mid n \geq 0\}$.

(b)

Restricted MS Word expressions give languages that are recognised by Finite Automata of the following form:

- States can be numbered $1, \ldots, k$, with all *forward* arcs going from a state to its successor (i.e., from state i to state $i + 1$).
- Backward arcs can go back any distance, but they must not "cross", in the sense that, if $i < j < k < l$, then you cannot have both the backward arcs $k \to i$ and $l \to j$.
- State 1 is the Start State, and state k is the only Final State.
- Loops must be labelled by all characters (to deal with the wildcard $*$).

Unrestricted MS Word expressions allow description of some non-regular languages, and in those cases there is no FA to recognise the language (by Kleene's Theorem).

5.

The extended regular expression (a*)b\1 matches the language $\{\mathbf{a}^n\mathbf{b}\mathbf{a}^n \mid n \geq 0\}$, but this is not regular, by the Pumping Lemma.

6.

If we drop 3(i),(ii), then no grouping or concatenation is possible.

So our regular expressions can be constructed from letters (and empty strings) just using alternatives and Kleene $*$. Because no grouping is possible, the Kleene $*$ can only be applied to a single letter. (Note that, if R is any regular expression, then $R^{**} = R^*$.) So in this case the only regular expressions we get are unions of expressions of the form x^* and y, where x, y are single letters from our alphabet. For example, we could have a* $\cup$ b. The only languages that match such expressions are those in which, firstly, no string has a mix of letters, and secondly, for any letter, we either have all strings consisting just of repetitions of that letter, or just the one-letter string with that letter alone.

For the Challenge: it would be a big task to investigate all these possibilities. We just give one, as an illustration.

Suppose we drop just 3(ii). So we forbid concatenation but allow grouping, alternatives and Kleene $*$.

Firstly, observe that, if R and S are regular expressions, then the regular expression $(R^* \cup S)^*$ may be simplified to $(R \cup S)^*$.

Proof:

($\Leftarrow$) This implication is clear, since R is a special case of R^*.

($\Rightarrow$) Let w be a string that matches $(R^* \cup S)^*$. We must show that it also matches $(R \cup S)^*$. If $w = \varepsilon$, then we are done, since the empty string matches $(R \cup S)^*$ using zero repetitions for the Kleene $*$. So assume $w \neq \varepsilon$. Then w may be partitioned into consecutive substrings, $w = w_1 \cdots w_k$, such that each w_i matches $R^* \cup S$. If w_i matches R^*, then it may be partitioned into consecutive substrings, $w_i = w_{i1} \cdots w_{il_i}$, such that each w_{ij} matches R. (Here, l_i is just the number of such substrings into which w_i is partitioned.) If, on the other hand, w_i matches S, then put $w_{11} = w$ and $l_i = 1$. Then, in any case, we see that we can partition w into strings w_{ij},

$$w = w_{11} \cdots w_{1l_1} w_{21} \cdots w_{2l_2} \ \cdots\cdots \ w_{k1} \cdots w_{kl_k},$$

such that each w_{ij} matches $R \cup S$. This establishes that w matches $(R \cup S)^*$.

So any string that matches $(R^* \cup S)^*$ must also match $(R \cup S)^*$.

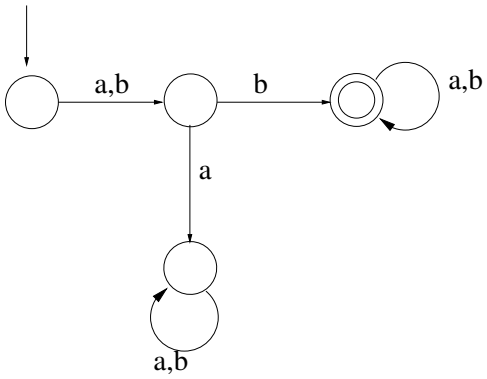
We have proved the implication in both directions. So, a string matches $(R^* \cup S)^*$ if and only if it matches $(R \cup S)^*$. Hence $(R^* \cup S)^*$ may be simplified to $(R \cup S)^*$. Q.E.D.

If the last operation applied in the expression is a Kleene $*$, then it has the form $(R_1 \cup \cdots \cup R_k)^*$. By the above observation, we may assume that none of the R_i has the Kleene $*$ as its last operation. We may also assume that none of them has grouping or $\cup$ as its last operation, since then we could just have listed the alternatives in the sequence $R_1, \ldots, R_k$. So the R_i must just be single letters or the empty string. Let L be the set of single letters that appear in the R_i. Then the expression is matched by any string which consists solely of letters from L, and by the empty string.

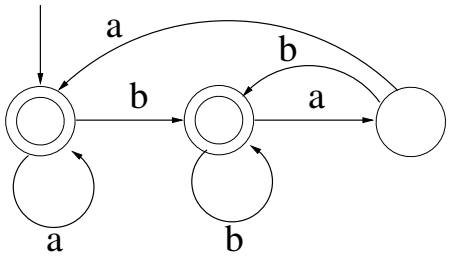
If the last operation applied in the expression is alternatives, say we have $R_1 \cup \cdots \cup R_k$, then we may suppose that each R_i is either the empty string or a single letter, or its last operation is the $*$. If the latter, then we are back in the situation of the previous paragraph.

In conclusion, the regular expressions that don't need concatenation in their construction are those formed from alternatives which are each either single letters or arbitrary repetitions of letters from some subset of the alphabet. (These subsets may be different for the different alternatives.)

7.



8.



9. (a) $\ell_{B,1} = 0$, $\ell_{W,1} = 0$, $\ell_{U,1} = 1$, $a_{B,1} = 1$, $a_{W,1} = 1$.
(b)

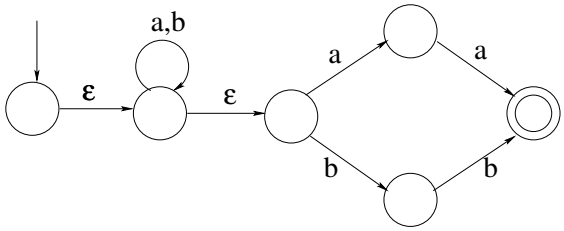
$$\begin{aligned}
\ell_{B,n+1} &= \ell_{B,n} + \ell_{U,n} \\
\ell_{W,n+1} &= \ell_{W,n} + \ell_{U,n} \\
\ell_{U,n+1} &= \ell_{B,n} + \ell_{W,n} + \ell_{U,n} + a_{B,n} + a_{W,n} \\
a_{B,n+1} &= \ell_{W,n} + a_{B,n} \\
a_{W,n+1} &= \ell_{B,n} + a_{W,n}
\end{aligned}$$

(c) Observe that, by symmetry, $\ell_{B,n} = \ell_{W,n}$ and $a_{B,n} = a_{W,n}$. So, if you want to work out the total number of legal positions on a path graph of some size, then it's enough to work out three quantities for each n from 1 up to the desired size: $\ell_{B,n}$ (equivalently, $\ell_{W,n}$); $\ell_{U,n}$; and $a_{B,n}$ (equivalently, $a_{W,n}$).
(d) Once you reach the desired value of n, the total number of legal positions is just $\ell_{B,n}+\ell_{W,n}+\ell_{U,n}$, which equals $2\ell_{B,n} + \ell_{U,n}$.

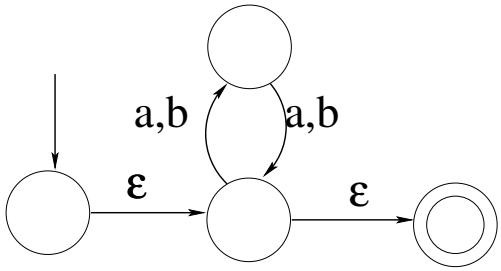
Note that $a_{B,n}$ and $a_{W,n}$ aren't part of this total. But you still need to use them, for smaller values of n, in the calculations leading up to the total.

10.

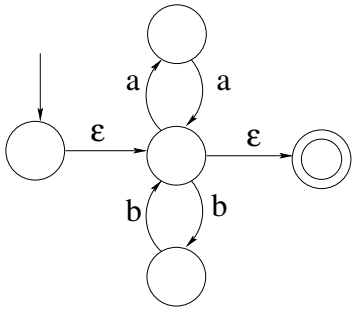
i)



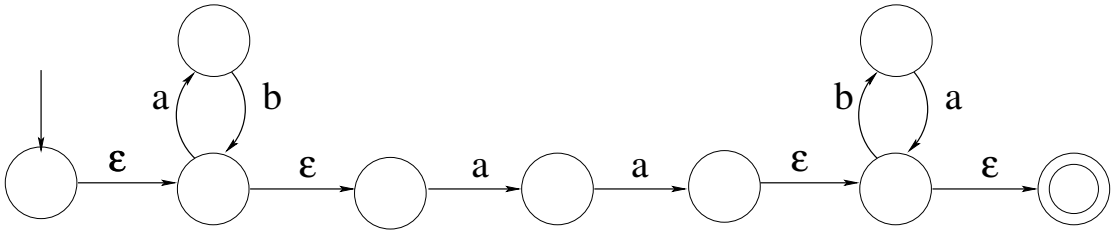
ii)



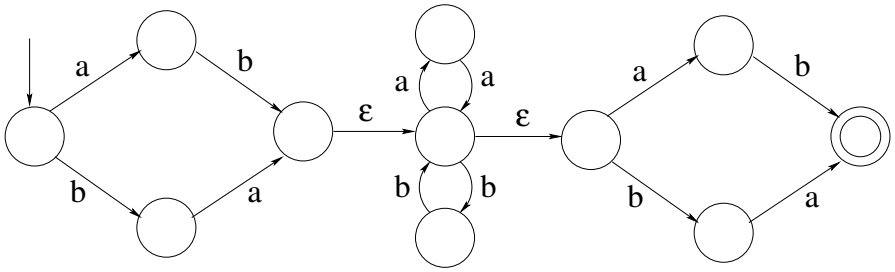
iii)



iv)

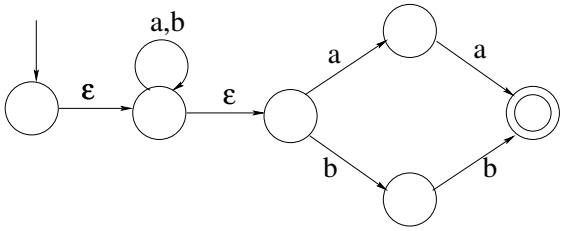


v)



11.

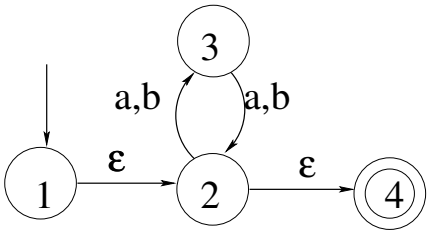
i) NFA



FA

	a	**b**
Start {1,2,3}	{2,3,4}	{2,5,3}
{2,3,4}	{2,3,4,6}	{2,3,5}
{2,3,5}	{2,3,4}	{2,3,5,6}
Final {2,3,4,6}	{2,3,4,6}	{2,3,5}
Final {2,3,5,6}	{2,3,4}	{2,3,5,6}

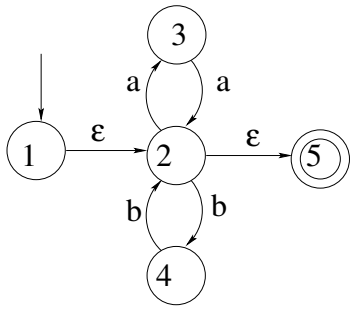
ii) NFA



FA

	a	**b**
Start, Final {1,2,4}	{3}	{3}
{3}	{2,4}	{2,4}
Final {2,4}	{3}	{3}

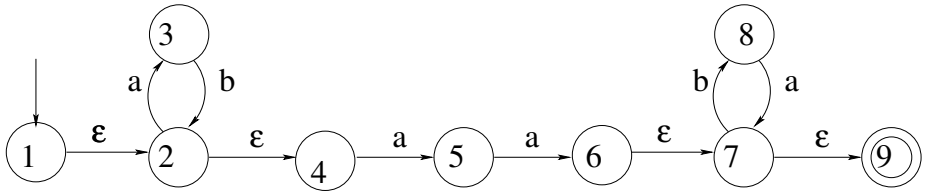
iii) NFA



FA

	a	b
Start {1,2,5}	{3}	{4}
{3}	{2,5}	ϕ
{4}	ϕ	{2,5}
Final {2,5}	{3}	{4}
ϕ	ϕ	ϕ

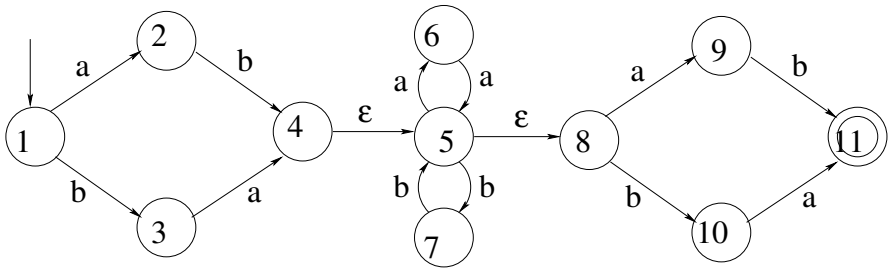
iv) NFA



FA

	a	b
Start {1,2,4}	{3,5}	ϕ
{3,5}	{6,7,9}	{2,4}
Final {6,7,9}	ϕ	{8}
{2,4}	{3,5}	ϕ
{8}	{7,9}	ϕ
Final {7,9}	ϕ	{8}
ϕ	ϕ	ϕ

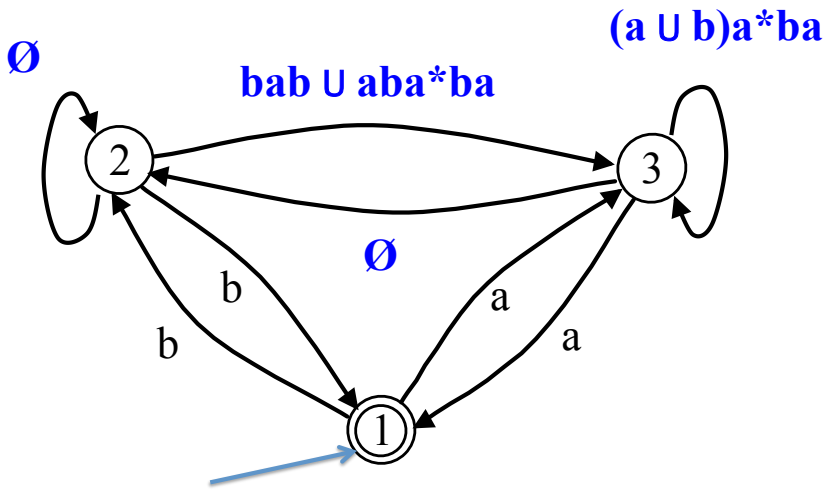
v) NFA



FA

	a	b
Start {1}	{2}	{3}
{2}	ϕ	{4,5,8}
{3}	{4,5,8}	ϕ
{4,5,8}	{6,9}	{7,10}
{6,9}	{5,8}	{11}
{7,10}	{11}	{5,8}
{5,8}	{6,9}	{7,10}
Final {11}	ϕ	ϕ
ϕ	ϕ	ϕ

12.



13.

Inductive basis:

When $n = 1$, the formula gives

$$\begin{aligned} F_1 &= \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right) - \left(\frac{1-\sqrt{5}}{2}\right)\right) \\ &= \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}-1+\sqrt{5}}{2}\right) \\ &= \frac{1}{\sqrt{5}} \cdot \frac{2\sqrt{5}}{2} \\ &= 1. \end{aligned}$$

When $n = 2$, the formula gives

$$\begin{aligned} F_2 &= \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^2 - \left(\frac{1-\sqrt{5}}{2}\right)^2\right) \\ &= \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right) + 1 - \left(\left(\frac{1-\sqrt{5}}{2}\right) + 1\right)\right) \\ &\quad \text{(using the fact that, for } x = (1 \pm \sqrt{5})/2, \text{ we know } x^2 - x - 1 = 0, \text{ i.e., } x^2 = x + 1) \\ &= \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2} - \frac{1-\sqrt{5}}{2} + 1 - 1\right) \\ &= \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}-1+\sqrt{5}}{2}\right) \\ &= \frac{1}{\sqrt{5}} \cdot \frac{2\sqrt{5}}{2} \\ &= 1. \end{aligned}$$

Since we have dealt with the cases $n \leq 2$, we may now assume $n \geq 3$.

Inductive step:

Suppose that, *for all* $m < n$, the formula holds for F_m. This is our inductive hypothesis. (It is a stronger type of inductive hypothesis than just assuming the formula holds for F_m when $m = n - 1$. But that's ok.)

For convenience, write

$$x = \frac{1+\sqrt{5}}{2}, \qquad y = \frac{1-\sqrt{5}}{2}.$$

Now consider F_n.

$$\begin{aligned} F_n &= F_{n-1} + F_{n-2} \\ &\quad \text{(using the definition of } F_n \text{ for } n \geq 3) \\ &= \frac{1}{\sqrt{5}}\left(x^{n-1} - y^{n-1}\right) + \frac{1}{\sqrt{5}}\left(x^{n-2} - y^{n-2}\right) \\ &\quad \text{(by our inductive hypothesis, applied twice: once with } m = n-1, \text{ and once with } m = n-2) \\ &= \frac{1}{\sqrt{5}}\left(x^{n-1} - y^{n-1} + x^{n-2} - y^{n-2}\right) \\ &= \frac{1}{\sqrt{5}}\left(x^{n-1} + x^{n-2} - (y^{n-1} + y^{n-2})\right) \end{aligned}$$

$$
\begin{aligned}
&= \frac{1}{\sqrt{5}}\left(x^{n-2}(x+1) - y^{n-2}(y+1)\right) \\
&= \frac{1}{\sqrt{5}}\left(x^{n-2}x^2 - y^{n-2}y^2\right) \\
&\quad \text{(using } x^2 = x+1 \text{ and } y^2 = y+1\text{)} \\
&= \frac{1}{\sqrt{5}}\left(x^n - y^n\right).
\end{aligned}
$$

So the formula holds for F_n as well.

In summary of the inductive step: we showed that, if the formula for F_m holds when $m < n$, then it holds for $m = n$.

Conclusion: by the Principle of Mathematical Induction, it follows that the formula is true for all values of n.

This exercise illustrates that, although it may be hard to discover the correct formula for some quantity (...surely, the expression for F_n comes as a surprise when you first meet it, and you may wonder how anyone came up with it!), *once you have the formula*, induction is a powerful tool for proving it. In research, we often *discover* a "fact" by an unpredictable process of creative exploration, and then *prove* it by a different technique — often, by induction.

This exercise has also established the connection between the famous Fibonacci numbers and the equally famous "Golden Ratio", $\varphi = (1+\sqrt{5})/2$.

Extra exercise to give more insight into this connection, for the curious or mathematically inclined: use the above result to prove that the ratio between two successive Fibonacci numbers tends to φ as $n \to \infty$.

Among the many applications of Fibonacci numbers in computer science is the fact that the Euclidean algorithm, for computing GCD, takes longest when the two input numbers are successive Fibonacci numbers.

14.

(a) states 2 and 4.

(b)

Base case ($n = 1$): We saw in (a) that `abba` can end up in State 4, which is the Final state. Therefore `abba` is accepted.

Inductive step: suppose $n \geq 2$, and that $(\texttt{abba})^{n-1}$ is accepted.
Since $(\texttt{abba})^{n-1}$ is accepted (by Inductive Hypothesis), there is some path it can take through the NFA that ends at the Final State. We also see that, if we are in the Final State, and we then read `abba`, then we can reach the Final State again, by following the path $4 \to 3 \to 2 \to 1 \to 4$. We can put these two paths together, to give a path for the string $(\texttt{abba})^{n-1}\texttt{abba}$ that goes from the Initial State to the Final State. Therefore this string is accepted by our NFA. But this string is just $(\texttt{abba})^n$. So $(\texttt{abba})^n$ is accepted.

Therefore, by the Principle of Mathematical Induction, the string $(\texttt{abba})^n$ is accepted for all $n \geq 1$.

Supplementary exercises

15. **aabaaa**, **aababb**, **aaabaa**, **aaabbb**, **bbbaaa**, **bbbabb**, **bbabaa**, **bbabbb**

16. **aaaaaa**, **aaaabb**, **aabbaa**, **aabbbb**, **bbaaaa**, **bbaabb**, **bbbbaa**, **bbbbbb**

17. A valid date not described is 5/3/2002, and invalid date described is 99/02/2002.

18. **H** : **M** : **S**, where $H = [0-9]|[01][0-9]|2[0-3]$, $M = [0-9]|[0-5][0-9]$, and $S = [0-9]|[0-5][0-9]$.

19. $(\text{-} \mid \ +)?(N| \ N.| \ N.N| \ .N)((e| \ E)(\text{-} \mid \ +)?N)?$, where $N = [0-9]^+$

20.

Preamble (which you can skip: to do so, go to 'Solutions' below):

The solutions given here use a particular file `/usr/share/dict/words` with 234,936 words, each on its own line. The file you use will quite likely be from a different source so the numbers may be somewhat different.

I decided to eliminate proper nouns first, for convenience, and put the result in `wordsFile`:

```
$ egrep '^[a-z]' /usr/share/dict/words > wordsFile
```

Here, the line starts with a prompt, which we have represented as `$` . The text in blue is entered by the user. The regular expression is between the two forward quotes (apostrophes). This regular expression matches text at the beginning of the line consisting of any lower-case letter of the alphabet. The `egrep` command picks any *line* containing a match for the regular expression and outputs all these lines. The " `> wordsFile`" causes all these lines to be put into the file `wordsFile`.

This file now has 210,679 words, one per line:

```
$ wc wordsFile
210679  210679 2249128 wordsFile
```

In my `/usr/share/dict/words`, there are no words with nonalphabetic characters, and the only upper-case characters occurred at the starts of words. So all the words in `wordsFile` now consist only of lower-case letters.

You may find that things are a bit different on your system. The words file may have words with apostrophes, like "don't", or hyphens. You can filter these out using `egrep` too, if you wish.

Solution:

Assume that your list of words is in a file called `wordsFile`, with each line consisting of one word (and nothing else).

Regular expression to match any vowel: [aeiou]

Regular expression to match any consonant: [^aeiou] , or [b-df-hj-np-tv-z]

Regular expression to match any word with no vowel: ^[^aeiou]+$

This uses the fact that, in `wordsFile`, each word goes from the start of the line (matched by ^) to the end of the line (matched by $). (In the more typical situation where words in a file are delimited by spaces, you could use [^aeiou]+ with a space before and after it.)

Words with no vowel:

```
$ egrep '^[^aeiou]+$' wordsFile
```

gives 132 such words.

Words with no vowel or 'y':

```
$ egrep '^[^aeiouy]+$' wordsFile
```

gives 30 such words, but some of these are single letters which are always counted as words, in their own right, in this list. We can exclude such, to find that there are ten such words in our file:

```
$ egrep '^[^aeiouy][^aeiouy]+$' wordsFile
cwm
grr
nth
```

```
pst
sh
st
tch
tck
th
tst
```

Words with no consonants:

```
$ egrep '^[aeiou]+$' wordsFile
a
aa
ae
ai
e
ea
eu
euouae
i
iao
ie
io
o
oe
oii
u
```

So there are 16 such words in this `wordsFile`.

Consonant-vowel alternations:

```
$ egrep '^[aeiou]?([^aeiou][aeiou])*[^aeiou]?$' wordsFile > consVowelAlt
$ wc consVowelAlt
13219   13219  103647 consVowelAlt
```

So there are 13,219 words in this file with an alternating consonant-vowel pattern. The fraction of words with this property, using this `wordsFile`, is 13219 / 210679 $\simeq$ 0.063, or 6.3%.

Longest run of consonants — and let's make it more interesting by treating 'y' as a vowel (though the solution is easily modified to treat it as a consonant):

```
$ egrep '[^aeiouy][^aeiouy][^aeiouy][^aeiouy][^aeiouy][^aeiouy]' wordsFile
archchronicler
bergschrund
fruchtschiefer
latchstring
lengthsman
postphthisic
veldtschoen
```

Adding an extra [^aeiouy] to the regular expression gives no solutions, so these are the words with the longest run of consonants with 'y' treated as a vowel (six).

Longest run of vowels (now treating 'y' as a consonant, again to focus on the extreme cases):

```
$ egrep '[aeiou][aeiou][aeiou][aeiou][aeiou][aeiou]' wordsFile
euouae
```

Again, the maximum is 6.

Further queries give a runner-up, `cadiueio`, and a 112-way tie for third place.

List of palindromes:

```
$ egrep '^([a-z]?)([a-z]?)([a-z]?)([a-z]?)([a-z]?)([a-z]?)([a-z]?)([a-z]?)([a-z]?)\9\8\7\6\5\4\3\2\1$'
wordsFile > palindromes
```

(all on a single line) gives a file, `palindromes`, of 135 palindromes. The longest have seven letters (e.g., `rotator`).

The expression used with `egrep` here was chosen to be long enough to cover any word that might be a palindrome in English, and so it turned out to be, for this word list. But the $\backslash n$ construct is usually only applicable when n is a *digit*, so $n \leq 9$. This means that this expression can be used to detect palindromes of up to $2 \times 9 + 1 = 19$ letters. If a word list had palindromes that were longer than this, then this approach would not detect them.

21. The answers for (a) and (b) are not unique.

(a) $((B^* \cup W^*)UU^*(B^* \cup W^*))^*$

(b) $((B^* \cup W^*)UU^*(B^* \cup W^*))^*(B^* \cup W^*)UU^*((BB^*WW^*) \cup (WW^*BB^*))$

(c) Yes. We saw in (c) that there is a Finite Automaton to recognise this language, so by Kleene's Theorem there must be a regular expression for it as well.

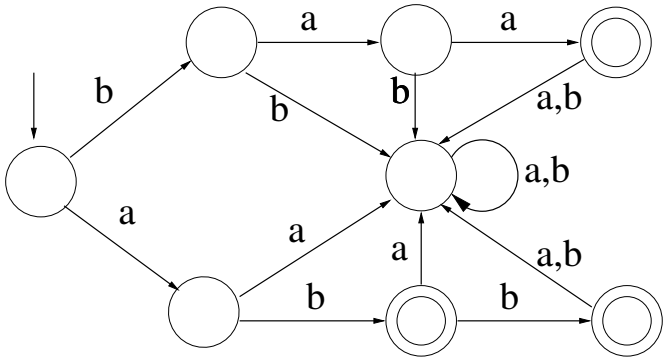
22.

1. All integers are arithmetic expressions.
2. If A and B are arithmetic expressions then so are: (A), A + B, A - B, A/B, and A*B.

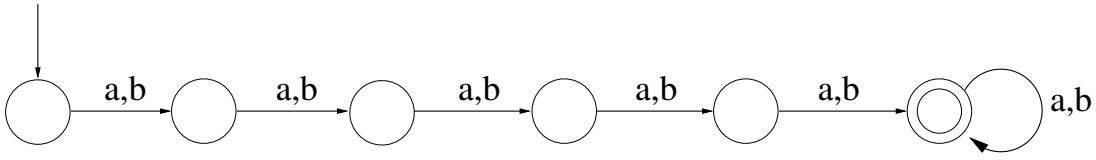
23.

1. The strings ϵ, **a**, and **b** are words in **PALINDROME**.
2. If S is a word in **PALINDROME** then so are: **a**S**a** and **b**S**b**.

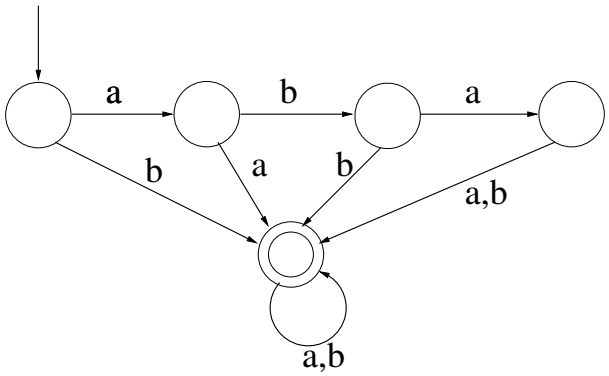
24.



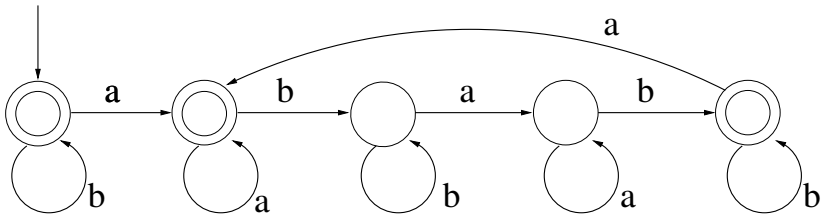
25.



26.



27.



28.

This uses the observation that a decimal number is divisible by 3 if and only if the sum of its digits is divisible by 3.[2] Furthermore, we may take each digit mod 3 when we do this.

If we ignored the ban on leading 0s, the following FA would do the job. In the table, State 0 serves as both Start State and the sole Final State.

state	transitions		
	0,3,6,9	1,4,7	2,5,8
0	0	1	2
1	1	2	0
2	2	0	1

How do we deal with leading 0s? Since they are forbidden, every string starting with a 0 should be rejected, regardless of what the subsequent letters are. Apart from this, 0 is treated like any other multiple of 3 (as in the above table). So we modify the above table to obtain the FA in the following table. This time, the Start State is denoted by −. The sole Final State is still the one labelled 0. The new state labelled 'x' is one from which you can never escape. It represents the unrecoverable error that occurs if a string starts with a 0.

state	transitions			
	0	3,6,9	1,4,7	2,5,8
−	x	0	1	2
x	x	x	x	x
0	0	0	1	2
1	1	1	2	0
2	2	2	0	1

29.

This exercise is best suited to students who are also doing (or have done) FIT2004, but the matters raised are still worth thinking about and looking up even if you haven't done that unit.

[2]This trick does not work for divisibility testing in general. But it does work for 9 as well as for 3.

If the transitions from a state are stored at that state (as in an adjacency list representation of a graph, for FIT2004 students), then it would be simplest to look at all transitions going out of a state p at the time that state is visited, rather than revisit p for each new symbol in the alphabet. So, inside the w-loop, it would be better to iterate over $p \in \text{endStates}(w)$, and for each such p, iterate over $x \in \{\text{a}, \text{b}\}$. This means interchanging the order of the second and third loops (in slide 24).

To deal with empty string transitions: consider the graph formed just from the empty string transitions, and form its *transitive closure* (see FIT2004, or most introductory books on algorithms). This tells you which node (state) is reachable from which other states using empty string transitions. This easily gives you, for each state, the set of all other states reachable from it by empty string transitions. This should all be done as *preprocessing*, before you start the algorithm given in lectures.

The (worst-case) complexity will be dominated by a factor of 2^n, where n is the number of states in the NFA, since the number of possible sets endStates(w) may be as large as 2^n, but no worse. The complexity will also have a polynomial factor, which will depend on the data structures used and the details of the algorithm.

The transitive closure can be computed in time $O(n^3)$ by the Floyd-Warshall algorithm. This is an additive contribution to the complexity (as it is pre-processing), so is swamped by the exponential term.

So the complexity is $O(2^n \text{poly}(n))$, for some polynomial poly(n). A more precise bound could be given, with more detailed study of the algorithm.

30.

	.	**-**	**[0-9]**
Start 1	2	3	4
2	6	6	5
3	2	6	4
Final 4	5	6	4
Final 5	6	6	5
6	6	6	6

31.

They all have the following minimum state finite automaton.

	a	**b**
Start/Final 1	1	1

32.

(i) No simplification possible.

(ii) Merge the two Final States (first and third rows).

(iii) No simplification possible.

(iv) Merge the two states {1,2,4} and {2,4}, and merge the two states {6,7,9} and {7,9}.

(v) Merge the two states {4,5,8} and {5,8}.

33.

We just do (iii) as the others involve long computations.

First, turn it into a GNFA by adding a new Final State 3 and an ε-transition from the Start State to it. The Start State is no longer a Final State.

Similarly, a new Start State is added, with a new empty string transition from it to the old Start State.

Then, applying the algorithm, we obtain (aa $\cup$ ab $\cup$ ba $\cup$ bb)*. This is equivalent to the original regular expression ((a $\cup$ b)(a $\cup$ b))*, and describes the language of all strings of even length.