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

FIT2014 Theory of Computation
Tutorial 4
Context-Free Grammars and the Pumping Lemma

SOLUTIONS

1.
(a)

(i). Z.
Note, in particular, that X is ***not*** the logical negation of W.

(ii). Y, Z

(b)

(i).

W: $\forall L\ (\mathbf{CFL}(L) \Rightarrow \mathbf{Infinite}(L))$

X: $\forall L\ (\mathbf{CFL}(L) \Rightarrow \neg\mathbf{Infinite}(L))$

Y: $\exists L\ (\mathbf{CFL}(L) \wedge \mathbf{Infinite}(L))$
Note: the following answer is incorrect: $\exists L\ (\mathbf{CFL}(L) \Rightarrow \mathbf{Infinite}(L))$. This is because the expression $\mathbf{CFL}(L) \Rightarrow \mathbf{Infinite}(L)$ is True whenever L is *not* context-free, as well as when L is context-free and infinite.

Z: $\exists L\ (\mathbf{CFL}(L) \wedge \neg\mathbf{Infinite}(L))$

(ii).

$$\begin{aligned}
\neg W &= \neg\forall L\ (\mathbf{CFL}(L) \Rightarrow \mathbf{Infinite}(L)) \\
&= \exists L\ \neg(\mathbf{CFL}(L) \Rightarrow \mathbf{Infinite}(L)) \\
&= \exists L\ \neg(\neg\mathbf{CFL}(L) \vee \mathbf{Infinite}(L)) \\
&\qquad \text{(since, in general, } A \Rightarrow B \text{ is equivalent to } \neg A \vee B) \\
&= \exists L\ (\neg\neg\mathbf{CFL}(L) \wedge \neg\mathbf{Infinite}(L)) \\
&\qquad \text{(by De Morgan's Law)} \\
&= \exists L\ (\mathbf{CFL}(L) \wedge \neg\mathbf{Infinite}(L))
\end{aligned}$$

(iii). Z.
Note, in particular, that X is ***not*** logically equivalent to $\neg W$.

2. Here is the grammar again, for convenience.

$$\begin{aligned} S &\to P && (1)\\ P &\to PP && (2)\\ P &\to \texttt{ha}Q && (3)\\ Q &\to Q\texttt{a} && (4)\\ Q &\to \varepsilon && (5) \end{aligned}$$

(a)

(i). Here is a derivation of the string `ha`.

$$\begin{aligned} S &\Rightarrow P && \text{(using rule (1))}\\ &\Rightarrow \texttt{ha}Q && \text{(using rule (3))}\\ &\Rightarrow \texttt{ha}\varepsilon && \text{(using rule (5))}\\ &= \texttt{ha} \end{aligned}$$

We now prove that this is a shortest string in LOL.

Any derivation (of any string) from S must start with the production $S \Rightarrow P$. So at least one of rules (2), (3) must be used in any derivation. If only (2) is used, then no symbol except P can ever be generated. So rule (3) must be used. This introduces two terminal symbols (in fact, the string `ha`). So every string in LOL must have *at least* two letters.

(In fact, we have shown that every string in LOL must contain the two-letter string `ha`. With this observation, we can say that `ha` is *the* shortest string in LOL, not just that it is *a* shortest string in LOL.)

(ii). The grammar is not regular, since not every production rule has, on its right-hand side, a semiword or a string of terminals. In particular, rules (2) and (4) are not of the required form.

(iii). LOL is regular language. It has the regular expression $\texttt{haa}^*(\texttt{haa}^*)^*$.

(iv). The grammar is not in Chomsky Normal Form. Apart from (2), the rules are not in the required form.

(b)

Step 1. How do we know such an x exists? In other words, how do we know LOL isn't empty?
Step 8. Consider: "... we can get strings that are even longer than that, and so on, indefinitely."

This needs formal justification. Don't "wave hands" to cover gaps in the reasoning, or rely on the reader to fill in gaps. This indicates a need for a proper proof by induction.

(c)

We prove this by induction on n.

Inductive basis ($n = 2$): we already know LOL has a string of length at least 2, since we saw in part (a)(i) that LOL contains the string `ha`.

Inductive step:

Suppose $n \geq 2$. As our Inductive Hypothesis, assume that LOL contains a string of length $\geq n$. Call it x. Since $x \in$ LOL, there is a derivation for x using the given grammar.

Now we just repeat the reasoning of Steps 3–7 in part (b).

This gives us a string y which is one letter longer than x. Since x has length $\geq n$, we deduce that y has length $\geq n + 1$. So LOL contains a string of length at least $n + 1$.

Therefore, by Mathematical Induction, for all $n \geq 2$, LOL contains a string of length $\geq n$.

(d)

This proof is correct, but unnecessarily complicated and long-winded.

The contradiction hinges on the fact that a *finite* language has a longest string, and therefore cannot have arbitrarily long strings. The deduction that LOL has arbitrarily long strings (Step 2) does indeed contradict the finiteness of LOL (assumed in Step 1), as stated in Step 3.

But you can derive the desired contradiction much more simply. All you need to do is to *use the (assumed) finiteness of LOL to drive the construction*, rather than "parking" the finiteness at the start of the proof and coming back to it at the very end. Here's how you do it.

Any finite language has a longest string; but we have already seen how to take any string in LOL and make another string in LOL that is one letter longer. You don't need to go to the extra trouble of showing that LOL has arbitrarily long strings.

(e)

Assume, by way of contradiction, that LOL is finite.

We know it is nonempty (part (a)(i)).

Since LOL is finite and nonempty, it must have a longest string. Call it x. *(Note how the assumed finiteness of LOL drives our construction of x.)*

Our earlier argument shows how to construct a string that is one letter longer than x. This contradicts the maximality of x.

Therefore our assumption, that LOL is finite, is incorrect.

Therefore LOL is infinite.

3.

(b)

Suppose L is context-free. Let k be the number of non-terminal symbols in a CFG for L. Let n be any positive integer such that $n^2 > 2^{k-1}$. Then $\mathbf{a}^{n^2} \in L$, so by the Pumping Lemma for Context-Free Languages, there exist strings u, v, x, y, z such that $w = uvxyz$, and the length of vxy is $\leq n$, and v, y are not both empty, and $uv^ixy^iz \in L$ for all $i \in \mathbb{N} \cup \{0\}$.

Let ℓ be the sum of the lengths of v and y. (Note, $\ell \geq 1$.)

From now on, the proof is very close to Tute 3, Q1(b).

Then the length of uv^ixy^iz is $n^2 + (i-1)\ell$. So the strings uv^ixy^iz have lengths $n^2, n^2+\ell, n^2+2\ell, n^2+3\ell, \ldots$. This is an infinite arithmetic sequence of numbers, with each consecutive pair being ℓ apart. But the sequence of lengths of strings in L is the sequence of square numbers, and by Tute 3, Q1(a), the gaps between them increase, eventually exceeding any specific number you care to name. So there comes a point where the gaps exceed ℓ, and some of the numbers $n^2 + (i-1)\ell$ fall between two squares. When that happens, $uv^ixy^iz \notin L$, a contradiction.

(c) This is mostly the same as the answer to Tute 3, Q1(c). The changed parts are underlined below.

Let L_1 be the language of all strings consisting entirely of 1s. This language is regular (since the regular expression 11* describes it).

Let L_2 be the language of binary string representations of adjacency matrices of graphs.

Assume L_2 is <u>context-free</u>. Then $L_1 \cap L_2$ is also <u>context-free</u>, since the <u>intersection of a context-free language and a regular language is context-free</u>.

But $L_1 \cap L_2$ is the language of all adjacency matrices consisting entirely of 1s. Such matrices always exist, for any n: they are the adjacency matrices of the complete graphs. (The *complete graph* on n vertices has every pair of vertices adjacent.) So $L_1 \cap L_2$ is actually the language L. But we have just shown in (b) that this is not <u>context-free</u>. So we have a contradiction.

Hence L_2 is not <u>context-free</u>.

4.

(a)

We prove that FootyScore is not regular.

Suppose FootyScore is regular. Then it has a Finite Automaton, by Kleene's Theorem. Let k be the number of states in a Finite Automaton for FootyScore. Let n be any positive integer such that $n > k$.

Let w be the string $(1^n, 1^n, 1^{7n})$. This may also be written

$$(\underbrace{11\cdots\cdots1}_{\text{goal-string, length } n}, \underbrace{11\cdots\cdots1}_{\text{behind-string, length } n}, \underbrace{11\cdots\cdots1}_{\text{point-string, length } 7n})$$

This string represents a score of a team that has kicked n goals and n behinds, giving $6n + n = 7n$ points. It satisfies the definition of strings in FootyScore, so it belongs to the language. Therefore, by the Pumping Lemma for Regular Languages, there exist strings x, y, z such that $w = xyz$, and the length of xy is $\leq n$, and y is non-empty, and $xy^iz \in$ FootyScore for all $i \geq 0$.

Firstly, observe that y cannot include either of the two commas, since if it did, repeating y (in forming xy^iz) would give more than two commas altogether, in violation of the definition of strings in FootyScore (which must have exactly two commas). Similarly, Y cannot contain either of the two parentheses.

Therefore y must fall entirely within the goal-string, or entirely within the behind-string, or entirely within the point-string. In each of these cases, repeating y would upset the score: it would change the number of goals, *or* the number of behinds, *or* the number of points, without changing either of the other two numbers. When only one of the three numbers g, b, p is changed, it is no longer true that $6g + b = p$, so the corresponding string no longer belongs to FootyScore.

This contradicts the conclusion from the Pumping Lemma, that $xy^iz \in$ FootyScore for all i. Hence our assumption, that FootyScore is regular, was wrong.

Therefore FootyScore is not regular.

The above proof did not use the fact that $|xy| \leq n$ (guaranteed by the Pumping Lemma). We could do a slightly different proof by using it. It tells us that the initial part xy of the string has length at most n, and since the goal string is more than n letters long, the substring xy must come before the first comma. This tells us that y lies entirely within the goal string, except that it may also contain the initial opening parenthesis (if x is empty, which is possible). In the former case, repeating y increases the number of goals without a corresponding increase in the number of points; in the latter case, the initial opening parenthesis is repeated. Either way, the rules of the language are violated, and again we can deduce that $xy^iz \notin$ FootyScore, so obtaining a contradiction.

This argument illustrates the main purpose of $|xy| \leq n$: it gives us more control over where y lies within w.

(b)

FootyScore is context-free, because it is generated by the following CFG.

$$S \rightarrow (G) \tag{6}$$
$$G \rightarrow \texttt{1}G\texttt{111111} \tag{7}$$
$$G \rightarrow \texttt{,}B \tag{8}$$
$$B \rightarrow \texttt{1}B\texttt{1} \tag{9}$$
$$B \rightarrow \texttt{,} \tag{10}$$

5.

First iteration: single letters.

`a` can be generated by the nonterminals A, D.

`b` can be generated by the nonterminal C.

Second iteration: pairs of consecutive letters (a.k.a. *digraphs*). We only need to consider digraphs that actually appear in the target string. In this case, we consider all digraphs except `aa`.

`ab`: The `a` can be generated by A or D, and the `b` can be generated by C, so the pair `ab` can come from either of the nonterminal pairs AC and DC. The pair AC can be produced by A, but the pair DC cannot be produced by a single nonterminal. So the sole nonterminal that can produce `ab` is A.

`bb`: This pair can come only from the pair CC, but this pair cannot come from a single nonterminal. So there is no nonterminal that can produce `bb`.

`ba`: This pair can come from either CA or CD. The former cannot come from a single nonterminal; the latter can come from B. So the only possibility here is B.

We summarise what we have worked out for the digraphs in the following table.

digraph	nonterminals that can produce it
`ab`	A
`bb`	—
`ba`	B

Third iteration: triples of consecutive letters (a.k.a. *trigraphs*). We only need to consider trigraphs that actually appear in the target string. For our string `abbba`, these are `abb`, `bbb`, `bba`.

We view each triple as a concatenation of two nonempty shorter strings.

`abb`: This can be formed as a concatenation of `a` followed by `bb`, or as a concatenation of `ab` followed by `b`. The former is not possible, because `bb` cannot be produced by a nonterminal (as we saw above). But for the latter, `ab` can be produced by A and `b` can be produced by C. So `abb` can be produced by AC. This in turn can be produced from the single nonterminal A. No other single nonterminal can do the job.

`bbb`: This can be formed as a concatenation of `b` followed by `bb`, or as a concatenation of `bb` followed by `b`. But in each case, the `bb` cannot be produced by any nonterminal. So it is also not possible to produce `bbb` from a nonterminal.

`bba`: This can be formed as a concatenation of `b` followed by `ba`, or as a concatenation of `bb` followed by `a`. We can rule the latter out, because of `bb`. For the former, it can be produced from CB, but that in turn cannot be produced by just a single nonterminal.

We summarise what we have found.

trigraph	nonterminals that can produce it
`abb`	A
`bbb`	—
`bba`	—

Fourth iteration: 4-tuples of consecutive letters (a.k.a. *tetragraphs*). For our string `abbba`, the ones we need to consider are `abbb`, `bbba`.

`abbb`: The only feasible split is `abb`,`b` which gives the nonterminal pair AC. This in turn can be produced from the single nonterminal A.

`bbba`: There is no feasible split. Each possible split includes a string (either the one before, or the one after, the split) that cannot be produced by a single nonterminal.

tetragraph	nonterminals that can produce it
`abbb`	A
`bbba`	—

Fifth and last iteration:

This is for our target string `abbba`.

The split `a`,`bbba` does not help, because `bbba` cannot be produced from a nonterminal. The split `ab`,`bba` is similarly unhelpful: see `bba` above. The split `abb`,`ba` can be produced by the nonterminal pair AB, which in turn can be produced only by the single nonterminal S. The split `abbb`,`a` can be produced by the nonterminal pairs AA and AD, but neither of these can come from a single nonterminal pair. So we are left with just S.

Since the starting nonterminal S is one of the nonterminals from which the target string can be produced, that string belongs to the language generated by the grammar.

6.

(a) Start with the given grammar:

$$\begin{array}{lcll} S & \to & P & (1) \\ P & \to & PP & (2) \\ P & \to & \texttt{ha}Q & (3) \\ Q & \to & Q\texttt{a} & (4) \\ Q & \to & \varepsilon & (5) \end{array}$$

First, eliminate the empty productions. There is just one here, $Q \to \varepsilon$. To ensure that we keep its effect, we create a new copy of every rule that has Q on its right-hand side, and delete that Q (i.e., replace it by the empty string; in effect, we apply the rule $Q \to \varepsilon$ to it). (It gets a bit more complicated for rules that have more than one Q on their right-hand side, but that doesn't happen here. Exercise: what should we do in that case?) In this case, there are two such rules, (3) and (4). So we get two new rules as well: $P \to \texttt{ha}$ and $Q \to \texttt{a}$. We now have the following grammar, equivalent to the original one:

$$\begin{array}{lcll} S & \to & P & (1) \\ P & \to & PP & (2) \\ P & \to & \texttt{ha}Q & (3) \\ P & \to & \texttt{ha} & \\ Q & \to & Q\texttt{a} & (4) \\ Q & \to & \texttt{a} & \end{array}$$

We now need rules to generate each terminal, by itself, *from a new nonterminal*. This gives new rules $A \to \texttt{a}$ and $H \to \texttt{h}$. (For `a`, it's not enough to rely on the rule $Q \to \texttt{a}$, which we already have, because nonterminal Q can be replaced by other things too; these new rules for the terminals need new symbols that will only ever be replaced by their corresponding terminals.) We obtain the following grammar.

$$\begin{array}{lcll} S & \to & P & (1) \\ P & \to & PP & (2) \\ P & \to & \texttt{ha}Q & (3) \\ P & \to & \texttt{ha} & \\ Q & \to & Q\texttt{a} & (4) \\ Q & \to & \texttt{a} & \\ A & \to & \texttt{a} & \\ H & \to & \texttt{h} & \end{array}$$

We then modify all rules with a terminal whose right-hand sides are not just a single terminal, replacing the terminals by the corresponding nonterminals we introduced above.

$$
\begin{array}{lcll}
S & \to & P & (1) \\
P & \to & PP & (2) \\
P & \to & HAQ & (3) \\
P & \to & HA & \\
Q & \to & QA & (4) \\
Q & \to & \texttt{a} & \\
A & \to & \texttt{a} & \\
H & \to & \texttt{h} &
\end{array}
$$

We must now eliminate unit productions. We just have one, namely (1), which changes S to P. It can be replaced by rules that replace S by anything that can be produced, by application of a single rule, from P. In this case, we have three rules for P. We therefore get three new rules, and the grammar is now as follows.

$$
\begin{array}{lcll}
S & \to & PP & \\
S & \to & HAQ & \\
S & \to & HA & \\
P & \to & PP & (2) \\
P & \to & HAQ & (3) \\
P & \to & HA & \\
Q & \to & QA & (4) \\
Q & \to & \texttt{a} & \\
A & \to & \texttt{a} & \\
H & \to & \texttt{h} &
\end{array}
$$

The last step is to deal with the rules whose right-hand sides have three or more nonterminals. We introduce the new nonterminal J and the new rule $J \to HA$, and use it to modify the grammar.

$$
\begin{array}{lcll}
S & \to & PP & \\
S & \to & JQ & \\
S & \to & HA & \\
P & \to & PP & (2) \\
P & \to & JQ & (3) \\
P & \to & HA & \\
Q & \to & QA & (4) \\
Q & \to & \texttt{a} & \\
A & \to & \texttt{a} & \\
H & \to & \texttt{h} & \\
J & \to & HA &
\end{array}
$$

(b)

Applying the CYK algorithm to this grammar and the target string `hahaa`, gives the following results at the successive iterations.

substring	nonterminals that can produce it	
h	H	
a	Q, A	
ha	S, P, J	... since these can all produce HA
ah	—	
aa	Q	... since $Q \Rightarrow QA \Rightarrow$ aa
hah	—	
aha	—	
haa	S, P	... using split ha,a and nonterminal pair JQ
haha	S, P	... using split ha,ha and nonterminal pair PP
ahaa	—	
hahaa	S, P	... using split ha,haa

Since S appears in the list of nonterminals that can produce hahaa, we conclude that hahaa belongs to the language generated by the grammar.

7.

We first prove by induction on n that, for every substring y of x, where $|y| \leq n$, the algorithm correctly finds all nonterminals in G that can generate y.

For the inductive basis, $n = 1$. The substrings y consist just of one letter each, and a grammar in Chomsky Normal Form will have, for each letter, some rules with that letter alone on their right-hand sides. The CYK algorithm finds all the nonterminals on the left of such rules, and specifies them as the nonterminals that can generate y, which is correct.

Inductive step: Let $n \geq 1$. Suppose the claim is true for substrings y of length $\leq n$. So, for each such substring, at the appropriate iteration, the algorithm finds all nonterminals that generate it.

Now consider what happens at the iteration that considers substrings of length $n + 1$. Let y be such a string. The algorithm considers each way of splitting y into two substrings, a left substring y_L and a right substring y_R. For each such split, we already know (from previous iterations) the set of nonterminals that can produce y_L and the set of nonterminals that can produce y_R. Since y_L and y_R are shorter than y, they each have $\leq n$ letters, so the inductive hypothesis applies, and tells us that these sets of nonterminals are complete and correct. The algorithm then constructs the set of all nonterminal pairs XY where X is in the first set of nonterminals (for y_L) and Y is in the second set of nonterminals (for y_R). For each such pair XY, it finds all rules of the form $W \to XY$, and gives these W as the nonterminals that can produce y.

This yields a set of nonterminals that can produce y. We need also to show that *every* nonterminal that can produce y must be in this set.

For y to be produced from a nonterminal symbol W, the first production must replace W by two other nonterminals, by the structure of the rules in Chomsky Normal Form (and using the fact that $|y| > 1$, so we cannot replace W by just a single terminal). Call these nonterminals X and Y. The rest of the production of y produces a left substring from X and a right substring from Y. (It is routine also to show that these two substrings cannot be empty. Just note that empty productions do not occur in CNF.) So, if y can be produced from W, then we can split it into two shorter substrings such that, for some nonterminals X and Y that produce the left and right substrings respectively, the grammar has the rule $W \to XY$. By the arguments in the previous paragraph, this shows that the algorithm does indeed find W among the nonterminals that can produce y.

All this works for any generated string y of length $n + 1$. This completes the inductive step.

Therefore, by Mathematical Induction, the claim is true for all n.

A string is generated by the CFG if and only if it can be produced from the Start symbol. We have just proved that the CYK algorithm correctly finds the list of all nonterminals that can produce a given string. Therefore, a string is generated by the CFG if and only if the set of nonterminals that the CYK algorithm finds for that string, as possible producers of it, includes the Start symbol.