KØBENHAVNS UNIVERSITET

Diskret Matematik og Formelle Sprog: Problem Set 5

Due: Sunday March 28 at 23:59 CET.
Submission: Please submit your solutions via *Absalon* as PDF file. State your name and e-mail address close to the top of the first page. Solutions should be written in LaTeX or some other math-aware typesetting system with reasonable margins on all sides (at least 2.5 cm). Please try to be precise and to the point in your solutions and refrain from vague statements. *Write so that a fellow student of yours can read, understand, and verify your solutions.* In addition to what is stated below, the general rules in the course information always apply.
Collaboration: Discussions of ideas in groups of two to three people are allowed—and indeed, encouraged—but you should always write up your solutions completely on your own, from start to finish, and you should understand all aspects of them fully. It is not allowed to compose draft solutions together and then continue editing individually, or to share any text, formulas, or pseudo-code. Also, no such material may be downloaded from the internet and/or used verbatim. Submitted solutions will be checked for plagiarism.
Grading: A score of 150 points is guaranteed to be enough to pass this problem set.
Please note: *If you have failed to pass some previous problem set(s), then running up enough extra points on this problem set to compensate for the total number of missing points from previous problem sets is guaranteed to make you eligible for the exam, so please make sure to work extra hard on the problems below in this applies to you.*
Questions: Please do not hesitate to ask the instructor or TAs if any problem statement is unclear, but please make sure to send private messages sometimes specific enough questions could give away the solution to your fellow students, and we want all of you to benefit from working on and learning on the problems. *Please note that you will need to watch both lectures from 2020 uploaded on Absalon, or to digest the material covered in those lectures in some other way, in order to be able to solve all problems on this problem set, so start by doing this before asking too many questions.* Good luck!

1 (120 p) In this problem, we want to understand the languages generated by specified regular expressions and context-free grammars.

1a (30 p) Which of the words below belong to the language generated by the regular expression $(b(ab)^* a) \mid (b^* ab^* ab^*)$? Motivate briefly your answers.

1. *bababaa*
2. *babbabbb*
3. *ba*
4. *babaabaaab*
5. *babbaabbbba*
6. *babba*

Solution: We give 5 p per correct answer. The regular expression accepts words that either have one b followed by arbitrary number of (ab) ending with a or that have two occurrences of a with an arbitrary number of b before, in between, or after. With this in mind, we get the following answers:

1. $bababa = b(ab)^3a$ ✓
2. $babbabbb = ba(b)^2a(b)^3$ ✓
3. $ba = b(ab)^0a$ ✓
4. *babaabaaab* ✗ (Letters a and b do not alternate, ruling out the first alternative, and for the second alternative there are too many occurrences of a.)
5. *babbaabbbba* ✗ (Same reason again.)
6. $babba = ba(b)^2a(b)^0$ ✓

1b (90 p) Consider the following context-free grammars, where a, b, c, d are terminals, A, B, S are non-terminals, and S is the starting symbol.

Grammar 1:

$$S \to abS \tag{1a}$$
$$S \to B \tag{1b}$$
$$B \to aB \tag{1c}$$
$$B \to cB \tag{1d}$$
$$B \to dB \tag{1e}$$
$$B \to \tag{1f}$$

Grammar 2:

$$S \to AbS \tag{2a}$$
$$S \to \tag{2b}$$
$$A \to aAa \tag{2c}$$
$$A \to \tag{2d}$$

Grammar 3:

$$S \to aSa \tag{3a}$$
$$S \to bS \tag{3b}$$
$$S \to \tag{3c}$$

Which of these grammars generate regular languages? For each grammar, write a regular expression that generates the same language, or argue why the language generated by the grammar is not regular.

Solution: We give 30 p per correct answer.

Grammar 1 generates strings that contain zero or more occurrences of ab follows by zero or more occurrences of (any alternation of) characters a, c, or d. This languages is captured by the regular expression $(ab)^*(a|c|d)^*$.

Grammar 2 generates zero or more substrings of the format where every substring consists of an even (possibly zero) number of occurrences of a followed by exactly one b. This corresponds to the regular expression $\big((aa)^*b\big)^*$.

The language generated by Grammar 3 is not regular. To see this, note that strings of the form a^nba^n are in the language while strings of the form a^nba^{n+1} are not. In order to

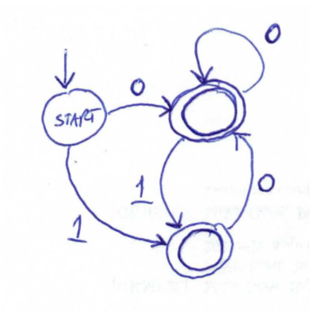


Figure 1: DFA proving regularity of language in Problem 2b.

distinguish between the two, the regular expression would need to do counting, but as mentioned in Mogesen's notes and during the lectures this is something that regular expressions cannot do. (You do not have to prove this to get full credits for this problem—referring to what is said in the notes or what we covered in class is enough.)

2 (120 p) In this problem, we want to write regular expressions and context-free grammars generating specified languages.

2a (30 p) Write a regular expression for the language consisting of all finite (possibly empty) bit strings (i.e., over the alphabet $\{0, 1\}$) that contain an even number of 0s. Examples of such strings are ε (the empty string), 00, 1010001, and 111, whereas 10 and 01010 do not qualify for membership.

Solution: The string can start with a sequence of 1s, leading up to the first 0, if any. If there is a 0, then it should be paired with another occurrence of 0, and in between them and after them we could have any number of 1s (including none). We can write this down as the regular expression $1^*(01^*01^*)^*$. (Other solutions are also possible, of course.)

2b (50 p) Write a regular expression for the language consisting of all finite, non-empty bit strings that contain no two consecutive 1s. Examples of strings in this language are 0000, 01010, and 1001, while 111, 00110, and ε do not qualify. For partial credit (if your regular expression is wrong or missing), argue why this language is clearly regular.

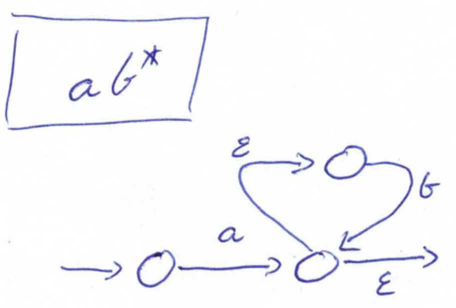
Solution: To see that this language is regular, it is sufficient to build a deterministic finite automaton that recognizes it, such as the DFA in Figure 1.

Suppose the string both starts and ends with a 0. Such strings are captured by the expression $00^*(100^*)^*$. To allow the string to also end in a 1, we can expand the expression slightly to $00^*(100^*)^*(1|\varepsilon)$.

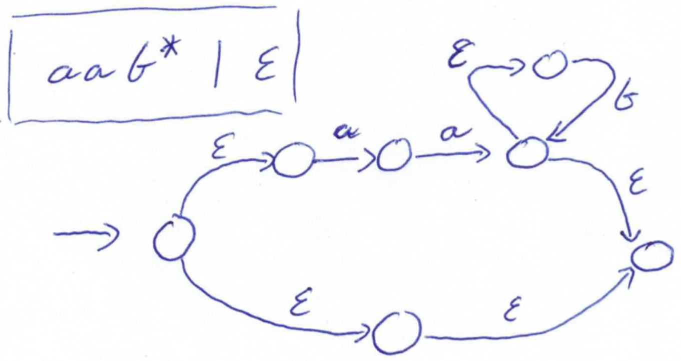
If the string instead starts and ends with a 1, we can describe it with the expression $1(00^*1)^*$. If we want to allow the string to end with 0 in this case, we can expand the expression to $1(00^*10^*)^*$.

Since the string will start with either 0 of 1, taking the alternative of these two expressions to get $\big(00^*(100^*)^*(1|\varepsilon)\big)\big|\big(1(00^*10^*)^*\big)$ is one possible solution.

We note that a much more elegant expression, which gets the language almost right, is $(1|\varepsilon)(00^*1)^*0^*$, but this regular expression also accepts the empty string, which is not allowed according to the description in the problem statement.



(a) NFA fragment for ab^*.



(b) NFA fragment for $aab^*|\varepsilon$.

Figure 2: NFA fragments in translation of regular expression in Problem 3.

2c (40 p) Give a context-free grammar for the language $\left\{a^m b^n c^{m+n} \,\middle|\, m, n \in \mathbb{N}\right\}$. Examples of strings in this language are $aacc$ and $abbccc$, whereas abc and $abccc$ do not make the cut.

Solution: Strings in this language should start with zero or more occurrences of a, with every occurrence being matched by a c at the end. At some point the string should switch to zero or more occurrences of b, again with every occurrence being matched by a c at the end, and no a can appear after the first b. (There is a slight ambiguity here in that we have to decide whether 0 is a natural number or not, but the example strings make clear that this is considered to be the case.)

A language as described above can be generated by, e.g., the following grammar:

$$
\begin{aligned}
&S \to aSc \\
&S \to B \\
&B \to bBc \\
&B \to \varepsilon
\end{aligned}
$$

3 (120+ p) Consider the regular expression $b^*(ab^*(aab^*|\varepsilon)cb^*)^*$

3a (60 p) Translate this regular expression to a nondeterministic finite automaton using the method from Mogensen's notes that we learned in class. Make sure to explain which part of the regular expression corresponds to which part of the NFA.

Solution: We follow the procedure outlined in Section 1.3 in Mogesen's notes. Two NFA fragments with corresponding regular (sub)expressions are shown in Figure 2. The full translation of the regular expression to a nondeterministic finite automaton is as in Figure 3, where we have added numbers to the states to be able to refer to them in the next subproblem.

3b (60 p) Translate the nondeterministic finite automaton to a deterministic finite automaton using the method from Mogensen's notes that we learned in class. Make sure to explain how you perform the subset construction, so that it is possible to follow your line of reasoning.

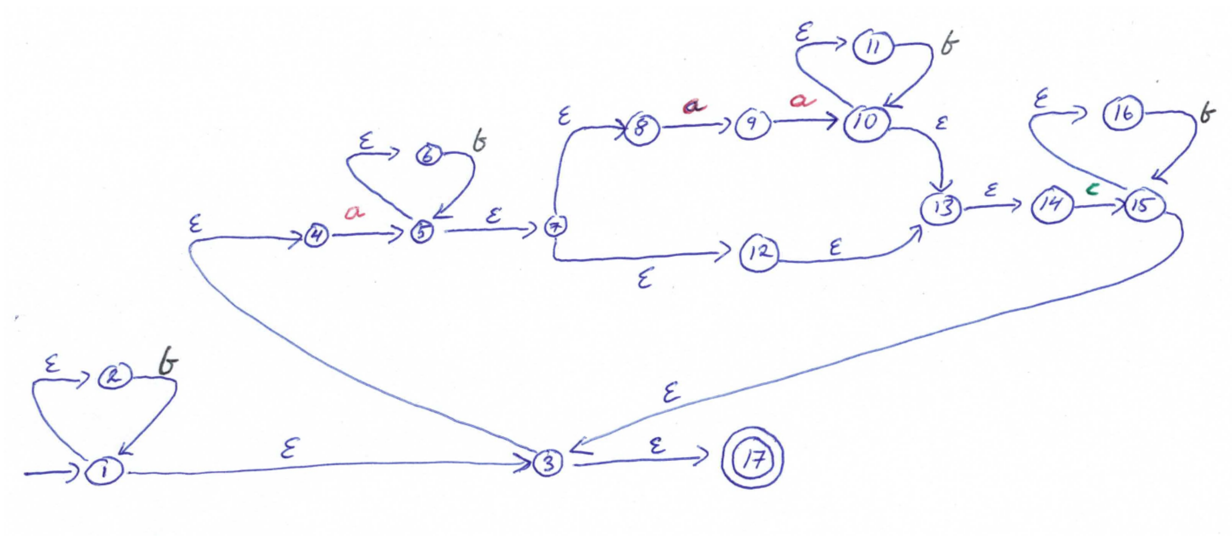


Figure 3: Full translation to NFA of regular expression in Problem 3.

Solution: We follow Algorithm 1.3 in Section 1.5.2 of Mogesen's notes. We will use calligraphic letters $\mathcal{A}$, $\mathcal{B}$, ... to refer to the constructed states in the deterministic finite automaton. Referring in what follows to the numbered states in the NFA in Figure 3, the starting state is

$$\varepsilon\text{-closure}(1) = \{1, 2, 3, 4, 17\} = \mathcal{A}$$

(where we recall that the ε-*closure* of a set of states S is any state that can be reached from some state in S via zero or more ε-transitions). The possible transitions from $\mathcal{A}$ on characters a, b, and c are

$$\begin{aligned} \text{move}(\mathcal{A}, a) &= \varepsilon\text{-closure}(5) = \{5, 6, 7, 8, 12, 13, 14\} = \mathcal{B} && [\textbf{new state}] \\ \text{move}(\mathcal{A}, b) &= \varepsilon\text{-closure}(1) = \mathcal{A} \\ \text{move}(\mathcal{A}, c) &= \emptyset && [\text{no transition on } c \text{ possible}] \end{aligned}$$

We consider next the new state $\mathcal{B}$, for which we get the transitions

$$\begin{aligned} \text{move}(\mathcal{B}, a) &= \varepsilon\text{-closure}(9) = \{9\} = \mathcal{C} && [\textbf{new state}] \\ \text{move}(\mathcal{B}, b) &= \varepsilon\text{-closure}(5) = \mathcal{B} \\ \text{move}(\mathcal{B}, c) &= \varepsilon\text{-closure}(15) = \{3, 4, 15, 16, 17\} = \mathcal{D} && [\textbf{new state}] \end{aligned}$$

Moving on to state $\mathcal{C}$ we obtain

$$\begin{aligned} \text{move}(\mathcal{C}, a) &= \varepsilon\text{-closure}(10) = \{10, 11, 13, 14\} = \mathcal{E} && [\textbf{new state}] \\ \text{move}(\mathcal{C}, b) &= \emptyset && [\text{no transition on } b \text{ possible}] \\ \text{move}(\mathcal{C}, c) &= \emptyset && [\text{no transition on } c \text{ possible}] \end{aligned}$$

and for state $\mathcal{D}$ we derive

$$\begin{aligned} \text{move}(\mathcal{D}, a) &= \varepsilon\text{-closure}(5) = \mathcal{B} \\ \text{move}(\mathcal{D}, b) &= \varepsilon\text{-closure}(15) = \mathcal{D} \\ \text{move}(\mathcal{D}, c) &= \emptyset && [\text{no transition on } c \text{ possible}] \end{aligned}$$

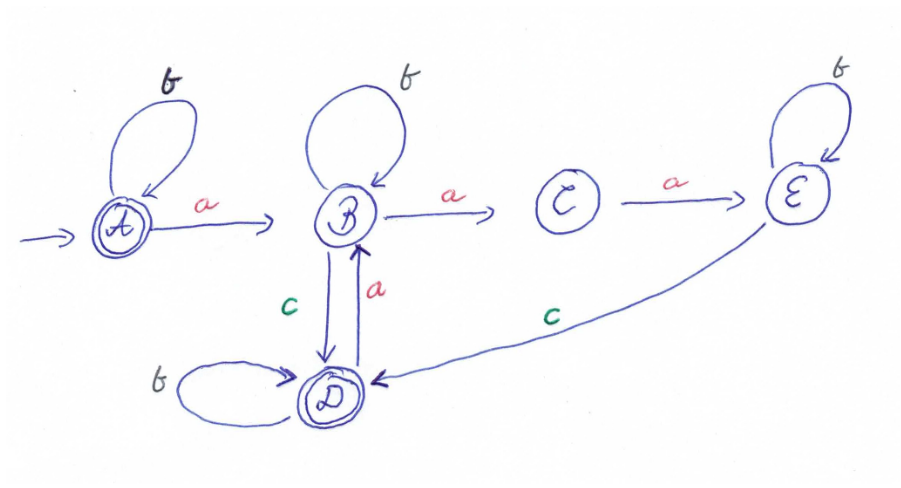


Figure 4: Conversion of NFA in Figure 3 to DFA.

Finally, for state $\mathcal{E}$ we have

$$\begin{aligned}
\text{move}(\mathcal{E}, a) &= \emptyset && \text{[no transition on } a \text{ possible]}\\
\text{move}(\mathcal{E}, b) &= \varepsilon\text{-closure}(10) = \mathcal{E}\\
\text{move}(\mathcal{E}, c) &= \varepsilon\text{-closure}(15) = \mathcal{D}
\end{aligned}$$

and since we have now considered all possible transitions from all states in our constructed DFA we are done. The new states containing the accepting state 17 in the NFA are $\mathcal{A}$ and $\mathcal{D}$, so these are the accepting states in our DFA. See Figure 4 for an illustration of the constructed automaton.

3c *Bonus problem (worth 20 p extra):* How small a DFA can you produce that accepts precisely the language generated by the regular expression above? (Note that that you can solve this problem even if you did not solve the other subproblems above.)

Solution: Looking at Figure 4, it is not hard to see that the states $\mathcal{A}$ and $\mathcal{D}$ have the same outgoing transitions. We can therefore merge these two states to get a smaller DFA as in Figure 5. Although we will not attempt a formal argument here, it is intuitively clear that this latter DFA is of optimal size. This is so since the automaton has to be able to count the number of occurrences of a that it has seen since the last occurrence of c, and this number can be in the range from 0 to 3.

We remark that it is not necessary here to go the formal route by first construction an NFA from the regular expression and then converting this NFA to a DFA. Just by studying the regular expression and understanding what it means, it is possible to write down the DFA in Figure 4 directly (and indeed, this was how the main instructor first did it).

4 (170 p) In this problem we want to construct parsers for context-free languages. We assume that (,), [,], +, *, and **num** are tokens return by a lexer (i.e., from the point of view of the parser these are terminals in the alphabet).

4a (70 p) Consider the following simplified version of the grammar for arithmetic expressions

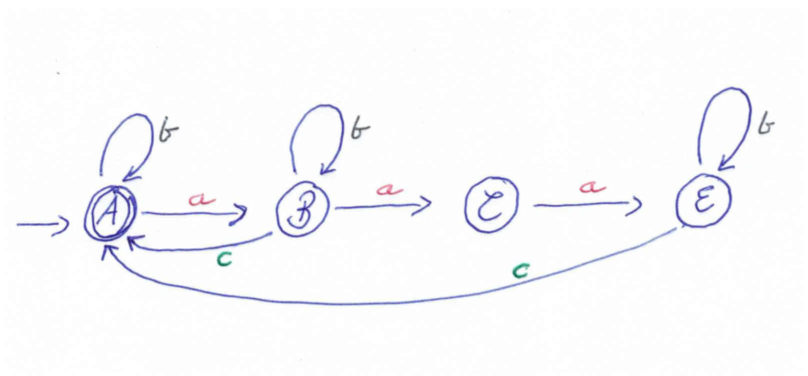


Figure 5: Smaller DFA equivalent to that in Figure 4.

with precedence that we saw in class, where E is the starting symbol:

$$E \to E + E_2 \tag{4a}$$
$$E \to E_2 \tag{4b}$$
$$E_2 \to E_2 * E_3 \tag{4c}$$
$$E_2 \to E_3 \tag{4d}$$
$$E_3 \to \mathbf{num} \tag{4e}$$
$$E_3 \to (\, E\,) \tag{4f}$$

Is this an LL(1) grammar?

If your answer is yes, then construct *FIRST*, *FOLLOW*, and *Nullable*, and give pseudo-code for a recursive descent parser.

If your answer is no, then explain why the grammar fails to be LL(1). Is it possible to build a predictive parser for the language in some other way by using more characters of look-ahead?

Solution: This grammar is not LL(1). Suppose that we are starting to parse the input and see a token **num**. Then with just this one token of look-ahead, there is no way of knowing whether we should use $E \to E + E_2$ or $E \to E_2$. This is just another way of saying that the *FIRST*-sets of these two productions are not disjoint.

Unfortunately, adding more characters of look-ahead does not help. With two characters of look-ahead we cannot know whether, e.g., (**num** should be parsed as $(\, E\,)$ or $(\, E\,) + E$, and if we add more characters of look-ahead, then we can also add extra parantheses to this problematic expression.

4b (100 p) Consider a grammar for parenthesized lists of **num** tokens as follows, with S as the starting symbol:

$$S \to P\ S \tag{5a}$$
$$S \to B\ S \tag{5b}$$
$$S \to N \tag{5c}$$
$$P \to (\ S\ ) \tag{5d}$$
$$B \to [\ S\ ] \tag{5e}$$
$$N \to \textbf{num}\ N \tag{5f}$$
$$N \to \tag{5g}$$

Is this an LL(1) grammar?

If your answer is yes, then construct *FIRST*, *FOLLOW*, and *Nullable*, and give pseudocode for a recursive descent parser.

If your answer is no, then explain why the grammar fails to be LL(1). Is it possible to build a predictive parser for the language in some other way by using more characters of look-ahead?

Solution: Getting straight to the point, this is an LL(1)-grammar. We show this by constructing *FIRST* and *Nullable* as in Section 2.7 and *FOLLOW* as in Section 2.9 in Mogesen's notes, after having added the new production $S' \to S\ \$$ to the grammar as also suggested in the notes, and then using this information to show that one token of look-ahead is sufficient for correct predictive parsing.

First, it is easy to verify that S and N are *Nullable*, as are the productions $S \to N$ and $N \to$, but that no other nonterminals or productions are *Nullable*. We get the following table for *FIRST* and *Nullable* (which is straightforward enough that we just write it down directly).

Production	*FIRST*	*Nullable*
$S' \to S\ \$$	(, [, **num**, \$	No
$S \to P\ S$	(	No
$S \to B\ S$	[	No
$S \to N$	**num**	Yes
$P \to (\ S\ )$	(	No
$B \to [\ S\ ]$	[	No
$N \to \textbf{num}\ N$	**num**	No
$N \to$	$\emptyset$	Yes

Using the *FIRST*-sets we just computed, we can extract the conditions for the *FOLLOW*-sets from the productions as listed below.

Production	Condition
$S' \to S\$$	$\{\$\} \subseteq FOLLOW(S)$
$S \to P\ S$	$\{(, [, \textbf{num}\} \subseteq FOLLOW(P)$, $FOLLOW(S) \subseteq FOLLOW(P)$
$S \to B\ S$	$\{(, [, \textbf{num}\} \subseteq FOLLOW(B)$, $FOLLOW(S) \subseteq FOLLOW(B)$
$S \to N$	$FOLLOW(S) \subseteq FOLLOW(N)$
$P \to (\ S\ )$	$\{)\} \subseteq FOLLOW(S)$
$B \to [\ S\ ]$	$\{]\} \subseteq FOLLOW(S)$
$N \to \textbf{num}\ N$	
$N \to$	

Applying the iterative procedure in Mogesen's notes, we compute the *FOLLOW*-sets for four iterations

Nonterminal	Iteration 1	Iteration 2	Iteration 3	Iteration 4
S	$\emptyset$	$),], \$$	$),], \$$	$),], \$$
P	$\emptyset$	$(, [, \textbf{num}$	$(, [, \textbf{num},),], \$$	$(, [, \textbf{num},),], \$$
B	$\emptyset$	$(, [, \textbf{num}$	$(, [, \textbf{num},),], \$$	$(, [, \textbf{num},),], \$$
N	$\emptyset$	$\emptyset$	$),], \$$	$),], \$$

until we reach the fixpoint.

We have now collected all the information needed to determine whether the grammar under study is an LL(1)-grammar or not. Consulting Mogensen's notes, we read that a given grammar is LL(1) precisely when for every nonterminal N it holds that it is correct to choose a production $N \to \alpha$ for the look-ahead c if

- $c \in \mathit{FIRST}(\alpha)$, or
- α is *Nullable* and $c \in \mathit{FOLLOW}(\alpha)$,

without ever having more than one possible production to choose from. (That is, it should never be the case that the condition above applies simultaneously for the same look-ahead token for two different productions $N \to \alpha_1$ and $N \to \alpha_2$.) We need to argue, based on the information that we have gathered so far, that this is true for the grammar we are considering.

If we expand the first table we build above with the *FOLLOW*-sets for the nullable productions, then we get

Production	*FIRST*	*Nullable*	*FOLLOW*
$S' \to S\$$	$(, [, \textbf{num}, \$$	No	
$S \to P\ S$	(	No	
$S \to B\ S$	[	No	
$S \to N$	**num**	Yes	$),], \$$
$P \to (\ S\ )$	(	No	
$B \to [\ S\ ]$	[	No	
$N \to \textbf{num}\ N$	**num**	No	
$N \to$	$\emptyset$	Yes	$),], \$$

From this table we can read off, among other things, the following interesting pieces of information:

- When starting to parse, if the first token is something other than $(, [, \textbf{num}$ (or end-of-file), then we have an error immediately.
- When parsing S and deciding on which production to apply, we should choose
  1. $S \to P\ S$ if the look-ahead is (;
  2. $S \to B\ S$ if the look-ahead is [;
  3. $S \to N$ if the look-ahead is one of $\textbf{num},),], \$$.
- When parsing N, we should choose
  1. $N \to \textbf{num}\ N$ if the look-ahead is **num**;
  2. $N \to$ if the look-ahead is one of $),], \$$.

- Regarding P and B, there can be no ambiguity since these nonterminals only have one production each.

In particular, we see that there can never be any conflict regarding which production to choose, but that one token of look-ahead is sufficient. This shows that we are indeed dealing with an LL(1)-grammar, just as claimed.

To turn an LL(1)-grammar into a recursive descent parser, the idea is that every nonterminal will correspond to a method/function/procedure, and the method for N will choose the appropriate production $N \to \alpha$, and then make calls in the right order to the (methods for the) nonterminals occurring in α and also parse any terminals/tokens. For more details, see the examples in Mogesen's notes or the handwritten lecture notes by the lecturer.