2.7 Exercises

2.7.1 Definition of regular languages

The definition of regular languages is Definition 2.1.

Exercise 2.1. Consider the alphabet $\Sigma = \{0, 1\}$. Using the inductive definition of regular languages, prove that the following languages are regular:

1. The set of words made of an arbitrary number of ones, followed by 01, followed by an arbitrary number of zeroes.

2. The set of odd binary numbers.

Exercise 2.2. Prove that any finite language is regular. Is the language $L = \{0^n 1^n \mid n \in \mathbb{N}\}$ regular? Give an intuition of why or why not.

Problem 2.3. Prove that, for all languages L and M: $(L^* M^*)^* = (L \cup M)^*$. Problem taken from Niwińsky and Rytter[15].

[15] Damian Niwińsky and Wojciech Rytter. *200 Problems in Formal Languages and Automata Theory*. University of Warsaw, 2017

2.7.2 Finite automata

We have defined the classes of finite automata in Section 2.3.

Exercise 2.4. For each of the following languages (defined on the alphabet $\Sigma = \{0, 1\}$), design a nondeterministic finite automaton (NFA) that accepts it:

1. the set of strings ending with 00;

2. the set of strings whose 3rd symbol, counted from the end of the string, is a 1;

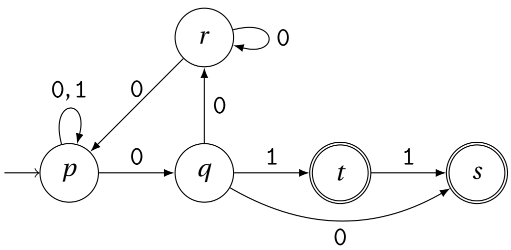
3. the set of strings where each pair of zeroes is directly followed by a pair of ones;

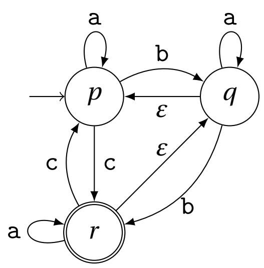
4. the set of strings not containing 101;

5. the set of binary numbers divisible by 4.

See the procedure for determinising automata in Section 2.4.2.

Exercise 2.5. Transform the following ε-NFA into DFA:





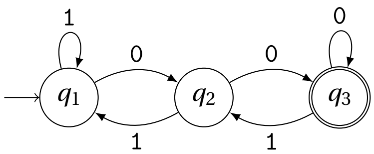
2.7.3 *Regular expressions*

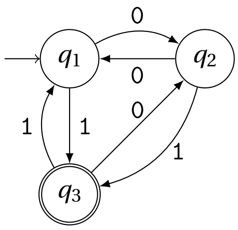
Exercise 2.6. Consider again all the languages from Exercise 2.4, and give a regular expression that define each of them.

The inductive definition of regular expressions is Definition 2.3.

A procedure to turn RE into ε-NFA has been given in Section 2.4.3.

Exercise 2.7. For each of the following DFA, give a regular expression accepting the same language:





A procedure to turn RE into ε-NFA has been given in Section 2.4.1.

Exercise 2.8. Convert the following RE into ε-NFA:

1. $\mathtt{01}^*$
2. $(\mathtt{0}+\mathtt{1})\mathtt{01}$
3. $\mathtt{00}(\mathtt{0}+\mathtt{1})^*$

2.7.4 *Extended regular expressions*

For the next exercises, you are asked to provide regular expressions using the 'extended regular expression' format (see Section 2.2.1), that is used in practice. You can test your answers using the regular expression library[16] `re` in Python, with its `re.search(pattern, string, flags=0)` method. The method receives an extended regular expression as the `pattern`, and returns a `Match object` indicating the first substring of `string` (if any) that matches the pattern. For example:

```
>>> import re
>>> re.search("(a|b|c)+","abcbcab")
<re.Match object; span=(0, 7), match='abcbcab'>
>>> re.search("(a|b|c)+","abcdef")
<re.Match object; span=(0, 3), match='abc'>
>>> re.search("(a|b|c)+","decbaf")
<re.Match object; span=(2, 5), match='cba'>
>>> re.search("(a|b|c)+","def")
```

[16] Python Software Foundation. re – Regular expression operations. https://docs.python.org/3/library/re.html. Online: accessed on April 12th, 2023

Observe that the last call returns nothing because no match was possible.

Exercise 2.9. Give an extended regular expression (ERE) that accepts any sequence of 5 characters, including the newline character `\n`.

Exercise 2.10. Give an ERE that accepts any string starting with an arbitrary number of \ followed by any number of $*$.

Exercise 2.11. UNIX-like shells (such as `bash`) allow the user to write *batch* files in which comments can be added. A line is defined to be a comment if it starts with a # sign. What ERE accepts such comments?

Exercise 2.12. Design an ERE that accepts numbers in scientific notation. Such a number must contain at least one digit and has two optional parts:

- a *decimal* part: a dot followed by a sequence of digits; and
- an *exponent* part: an `E` followed by an integer that may be prefixed by `+` or `-`.

For example, the following strings are valid numbers in scientific notation: `42`, `66.4E-5`, `8E17`

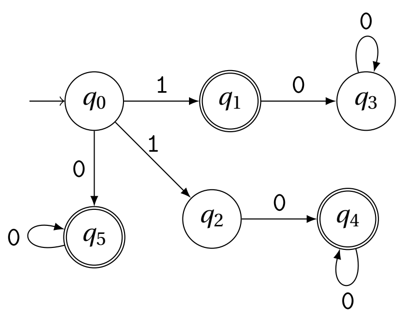
Exercise 2.13. Design an ERE that accepts 'correct' sentences that fulfill the following criteria: (i) no prepending/appending spaces; (ii) the first word must start with a capital letter; (iii) the sentence must end with a dot `.`; (iv) the phrase must be made of one or more words (made of the characters `a...z` and `A...Z`) separated by a single space; (v) there must be one sentence per line; and (vi) punctuation signs other than the dot are not allowed.

Exercise 2.14. Give an ERE that accepts old school DOS-style filenames respecting the following criteria. First, each filename starts by 8 characters (among `a...z`, `A...Z` and `_`), and the first five characters must be `abcde`. Next, each filename has an extension which is `.ext`. Finally, the ERE must accept accept the filename *only* (i.e., without the extension)!

For example, on `abcdeLOL.ext`, the ERE must accept `abcdeLOL`.

2.7.5 *Minimisation of automata and other operations*

Exercise 2.15. Here is a finite automaton:



We want to compute a *minimal* DFA that accepts the same language as this automaton:

The subset construction technique is the method of Section 2.4.2.

1. First, determinise this automaton using the subset construction technique.
2. Is the resulting automaton *minimal* (in terms of number of states)?
3. For each state q of the resulting DFA, give, as a regular expression, the language L_q that the automaton would accept if q were the initial state.

4. Based on the information computed at the previous point, propose a smaller DFA accepting the same language as the original automaton.

5. Finally, apply the systematic method to minimise DFA and compare the results.

The method to minimise DFA is in Section 2.5, Algorithm 1.

Exercise 2.16. Consider again the finite automaton at the beginning of Exercise 2.15. Let us denote this automaton by $A = \langle Q, \Sigma, \delta, q_0, F \rangle$. Then:

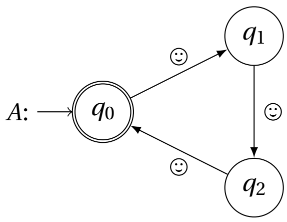
1. draw the automaton $A' = \langle Q, \Sigma, \delta, q_0, Q \setminus F \rangle$. How can you describe the relationship between A and A' in plain english?

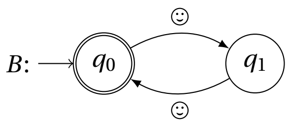
2. what is the relationship between $L(A)$ and $L(A')$? Do they have a non-empty intersection? Is it the case that $L(A)$ is the complement of $L(A')$?

3. . Apply the systematic method to compute the complement of a finite automaton, and check that the result indeed accepts the complement of $L(A)$.

The method to complement a finite automaton is found in Section 2.6

Exercise 2.17. Here are two finite automata A and B on the singleton alphabet {☺}:





1. Describe, in plain english, the respective languages of these automata. Then, describe, again in plain english, the intersection of these two languages.

2. Use the method to compute the intersection of two finite automata on A and B, and check whether the result matches your intuition.

The algorithm to compute the intersection is in Section 2.6 again.

2.7.6 *The scanner generator JFlex*

For this part of the exercises, we will rely on the scanner generator JFlex. A *scanner* is a program that reads text on the standard input and prints it on standard output after applying operations. For example, a filter that replaces all `a` with `b` and that receives `abracadabra` on input would output `bbrbcbdbbrb`. Then, JFlex is a tool that generates such a scanner based on a set of regular expressions that specify which part of the input should be matched and modified. To recognise these regular expressions, JFlex is based on the theory of finite automata that we have studied. The generated scanned is, in fact, a Java function.

JFlex can be dowloaded from `http://jflex.de` and a user manual is at `http://jflex.de/manual.html`.

We start by a very short explanation of the tool.

Specification format A JFlex specification is made of three parts separated by lines with `%%`:

1. the first part is the user code. It can contain any Java code, that will be added at the beginning of the generated scanner.

2. the second part contains options and declarations.

   The options include:

   - `%class Name` to tell JFlex to produce a scanner inside the classe called `Name`;
   - `%unicode` to enable unicode input;
   - `%line` and `%column` to enable line and column counting respectively;
   - `%standalone` to generate a scanner that is not called by a parser.

   Then, some extra Java code included between `%{` and `%}` can be generated. It will be copied verbatim *inside* the generated Java class (contrary to the code of the first part which appears outside of the class).

   Finally, some ERE can be defined. They will be used as macros in part 3 of the file to enhance readability. For example:

```
Comment    = "/*" [^*] ~"*/" | "/*" "*"+ "/"
```

   defines the macro `Comment` and associates it to the given ERE.

3. the third part contains the core of the scanner. It is a series of rules that associate actions (in terms of Java code) to the regular expressions. Each rule is of the form:

```
Regex    {Action}
```

   where:

   - `Regex` is an *extended regular expression* (ERE), that can use some of the regular expressions defined in part 2 as macros (using curly braces around their names, for example: `{Comment}`);
   - `Action` is a *Java code snippet* that will be executed each time a *token* matching `Regex` is found.

   For example, the rule:

```
"=="        { return symbol(sym.EQEQ); }
```

   instructs the scanner to return `sym.EQEQ` every time == is found on the input.

The reader is advised to have a look at the JFlex documentation[17] for a comprehensive example

[17] Gerwin Klein, Steve Rowe, and Régis Décamps. Jflex user's manual. https://jflex.de/manual.html, March 2023. Version 1.9.1. Online: accessed on April, 12th, 2023

Variables and special actions When writing *actions*, some special variables and macros can be accessed:

- `yylength()` contains the *length* of the recognized token
- `yytext()` is a the actual string that was matched by the regular expression.

- `yyline` is the line counter (requires the option %line).
- `yycolumn` is the column counter (requires the option %column).
- `EOF` is the End-of-file marker .

On Mac, files which do not end with an empty line can make the lexer "forget about" the last line. When you test your lexer, make sure that your test files end with an empty line.

Meta states In order to track the progress of the scanner, *states* can be used. Each action can change or query the current state of the scanner. This amounts to having a finite automaton running in parallel to the scanner. This way, when one particular token is recognised, the scanner can change the current state, which allows to 'store' some information that can be checked during a further call to the scanner.

There are actually two kind of states:

- inclusive states, declared by `%state`, which acts as Booleans (the scanner can be in several of those states at a time);
- exclusive states, declared by `%xstate` which are mutually exclusive (like regular automata states).

Then, the rules can be associated to states, and are active only in these states. A state can be 'activated' using the function `yybegin(S)` in the code (where `S` is the name of the state to activate). Here is an example:

```
%%
xstate YYINITIAL, PRINT;
%%
<YYINITIAL> {
   "print" {yybegin(PRINT);}
}
<PRINT> {
   ";" {yybegin(YYINITIAL);}
   .   {System.out.println(yytext());}
}
```

Executable To obtain the scanner executable, follow these steps[18]:

1. Generate the scanner code with:

   ```
   java -jar jflex-1.9.1.jar myspec.flex+
   ```

   which creates the file `Lexer.java` containing the `Lexer` class (the `%class` option can be used to change this);
2. compile the code into a class file: `javac Lexer.java` which creates `Lexer.class`;
3. run it with `java Lexer inputfile`.

Here are now some exercises to get you familiar with JFlex:

Exercise 2.18. Write a scanner that outputs its input file with line numbers in front of every line.

[18] Assuming you are using version 1.9.1, which is the last version at the time of writing.

Exercise 2.19. Write a scanner that outputs the number of alphanumeric characters, alphanumeric words and alphanumeric lines in the input file.

Exercise 2.20. Write a scanner that only shows the content of comments in the input file. Such comments are enclosed within curly braces `{ }`. You can assume that the input file does not contain curly braces inside comments.

Exercise 2.21. Write a scanner that transforms the input text as follows. It should replace the word `compiler` by `nope` if the line starts with an `a`; by `???` if it starts with a `b` and by `!!!` if it starts with a `c`.

Exercise 2.22. Write a *lexical analysis function* that recognises the following *tokens*:

- decimal numbers in scientific notation (e.g. `-0.4E-1`);
- C99 variable identifiers: they start with an an alphabetical symbol, followed by an arbitrary number of alphanumeric symbols or underscores;
- relational operators (<, >, ==, `!=`, >=, <=, `!`)
- The keywords `if`, `then` and `else`.

Each call to the function must seek the next token on the input. Every time a token is found, your function must output a message of the form `TOKEN NAME: token` (for example: `C99VAR: myvariable`) and return an object `Symbol` containing the token type, its value and its position (line and column). Templates for the `Symbol` and `LexicalUnit` classes are provided on the *Université Virtuelle*.

3.4 Exercises

3.4.1 Context-free languages

Exercise 3.1. Is the language $L = \{1^n \mid \exists m \in \mathbb{N} : n = m^2\}$ regular? Prove your answer using the techniques we have used at the beginning of the chapter.

3.4.2 Grammars

Exercise 3.2. Informally describe the languages generated by the following grammars and specify the classes of the Chomsky hierarchy they belong to:

(1)	S	$\rightarrow$	0
(2)		$\rightarrow$	1
(3)		$\rightarrow$	1S

(1)	S	$\rightarrow$	a
(2)		$\rightarrow$	$*SS$
(3)		$\rightarrow$	$+SS$

(1)	S	$\rightarrow$	abc$A$
(2)		$\rightarrow$	Aabc
(3)	A	$\rightarrow$	ε
(4)	Aa	$\rightarrow$	Sa
(5)	cA	$\rightarrow$	cS

Then, give a derivation of the word 1110 according to the first grammar; a derivation of the word $*$ + a + aa $*$ aa according to the second grammar G_2 and a derivation of the word *abcabc* produced by grammar G_3.

Exercise 3.3. Consider the following grammar:

(1)	S	$\rightarrow$	AB
(2)	A	$\rightarrow$	Aa
(3)	A	$\rightarrow$	bB
(4)	B	$\rightarrow$	a
(5)	B	$\rightarrow$	Sb

1. To which classes of the Chomsky hierarchy does this grammar belong?

2. Give the derivation trees of the three following sentential forms:

   (a) ba$S$b;

   (b) bBABb;

   (c) baabaab.

3. Give the *leftmost* and *rightmost* derivations for baabaab.

Exercise 3.4. Give a *context-free* grammar that generates all strings of a and b (in any order) such that there are strictly more a than b. Test your grammar on the input baaba by giving a derivation on this word.

Exercise 3.5. Give a *context-free* grammar that generates the language:

$$\{\mathtt{a}^n\mathtt{b}^m\mathtt{c}^\ell \mid n + m = \ell\}$$

Exercise 3.6. Give a context-sensitive grammar for the language

$$\{\mathtt{a}^m\mathtt{b}^n\mathtt{c}^m\mathtt{d}^n \mid m \geq 1, n \geq 1\}.$$

Do you think such a language can be generated by a context-free grammar? Explain why.

Exercise 3.7. Give a context-free grammar that generates all the arithmetic expressions on the alphabet $\{(,),+,.,0,1\}$ that evaluate to 2. Problem taken from Niwiński and Rytter[9].

Hint: start by generating all expressions that evaluate to 0, then to 1, then to 2.

[9] Damian Niwiński and Wojciech Rytter. *200 Problems in Formal Languages and Automata Theory*. University of Warsaw, 2017

4.5 Exercises

4.5.1 Pushdown automata

Exercise 4.1. Give a PDA that accepts the language containing all words of the form ww^R where w is any given word on the alphabet $\Sigma = \{\texttt{a}, \texttt{b}\}$ and w^R is the mirror image of w. Test your automaton on the input word `abaaaaba`, by giving an accepting run of your automaton on this word. Does your automaton accept by empty stack or by accepting state?

Exercise 4.2. (Exam question in 2014) Give the diagram of a deterministic pushdown automaton, on the alphabet $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, \texttt{e}\}$,that accepts the language $L = \{(\texttt{ab})^n\texttt{c}(\texttt{de})^n \mid n \geq 0\}$ using the empty stack acceptance condition.

4.5.2 Grammar transformations

Exercise 4.3. Remove the useless symbols in the following grammars:

The techniques to do so have been described in Section 4.4.3.

(1)	S	$\rightarrow$	$\texttt{a}$
(2)		$\rightarrow$	A
(3)	A	$\rightarrow$	AB
(4)	B	$\rightarrow$	$\texttt{b}$

(1)	S	$\rightarrow$	A
(2)		$\rightarrow$	B
(3)	A	$\rightarrow$	$\texttt{a}B$
(4)		$\rightarrow$	$\texttt{b}S$
(5)		$\rightarrow$	$\texttt{b}$
(6)	B	$\rightarrow$	AB
(7)		$\rightarrow$	$B\texttt{a}$
(8)	C	$\rightarrow$	AS
(9)		$\rightarrow$	$\texttt{b}$

Exercise 4.4. Consider the following grammar:

(1)	E	$\rightarrow$	$E\ op\ E$
(2)		$\rightarrow$	$\texttt{ID}[E]$
(3)		$\rightarrow$	$\texttt{ID}$
(4)	op	$\rightarrow$	$*$
(5)		$\rightarrow$	$/$
(6)		$\rightarrow$	$+$
(7)		$\rightarrow$	$-$
(8)		$\rightarrow$	$\Rightarrow$

1. Show that it is *ambiguous*.

See Definition 4.5 and Section 4.4.4.

2. The priorities of the various operators are as follows: [] and $\Rightarrow$ have higher priority than $*$ and $/$, which have higher priority than $+$ and $-$. Modify the grammar to take operator precedence into account as well as left associativity.

See Section 4.4 for the techniques.

Exercise 4.5. Left-factor the following production rules:

(1)	stmt	$\rightarrow$	`if` expr `then` stmt – list `end if`
(2)	stmt	$\rightarrow$	`if` expr `then` stmt – list `else` stmt – list `end if`

See Section 4.4 for the techniques.

Exercise 4.6. Apply the left recursion removal algorithm to the following grammar:

(1)	E	$\rightarrow$	$E+T$
(2)		$\rightarrow$	T
(3)	T	$\rightarrow$	$T*P$
(4)		$\rightarrow$	P
(5)	P	$\rightarrow$	$\texttt{ID}$

Exercise 4.7. (Excerpt from an exam question) Remove unproductive symbols and then inaccessible symbols from the following grammar:

(1)	S	$\rightarrow$	$\texttt{a}E$
(2)		$\rightarrow$	$\texttt{b}F$
(3)	E	$\rightarrow$	$\texttt{b}E$
(4)		$\rightarrow$	ε
(5)	F	$\rightarrow$	$\texttt{a}F$
(6)		$\rightarrow$	$\texttt{a}G$
(7)		$\rightarrow$	$\texttt{a}HD$
(8)	G	$\rightarrow$	$G\texttt{c}$
(9)		$\rightarrow$	d
(10)	H	$\rightarrow$	$C\texttt{a}$
(11)	C	$\rightarrow$	$H\texttt{b}$
(12)	D	$\rightarrow$	$\texttt{ab}$

Then, remove left-recursion and perform left-factoring whenever possible.

5.7 Exercises

5.7.1 First and Follow sets

(1)	<S>	→	<program>
(2)	<program>	→	`begin` <statement list>
(3)	<statement list>	→	<statement> <statement tail>
(4)	<statement tail>	→	<statement> <statement tail>
(5)	<statement tail>	→	ε
(6)	<statement>	→	`ID :=` <expression> `;`
(7)	<statement>	→	`read (` <id list> `) ;`
(8)	<statement>	→	`write (` <expr list> `) ;`
(9)	<id list>	→	`ID` <id tail>
(10)	<id tail>	→	`, ID` <id tail>
(11)	<id tail>	→	ε
(12)	<expr list>	→	<expression> <expr tail>
(13)	<expr tail>	→	`,` <expression> <expr tail>
(14)	<expr tail>	→	ε
(15)	<expression>	→	<primary> <primary tail>
(16)	<primary tail>	→	<add op> <primary> <primary tail>
(17)	<primary tail>	→	ε
(18)	<primary>	→	(<expression>)
(19)	<primary>	→	`ID`
(20)	<primary>	→	`INTLIT`
(21)	<add op>	→	+
(22)	<add op>	→	−

Exercise 5.1. We consider the grammar given above.

1. Give the values of $\mathrm{First}^1(A)$ and the $\mathrm{Follow}^1(A)$ sets for all variables A of the grammar.
2. Give the values of First^2(<expression>) and Follow^2(<expression>).

5.7.2 LL(k) grammars

Exercise 5.2. Consider the four grammars in Figure 5.8. Which of those grammars are LL(1)? Justify your answers.

Exercise 5.3. Give the LL(1) action table for the following grammar:

(1)	<S>	→	<expr>$
(2)	<expr>	→	− <expr>
(3)	<expr>	→	(<expr>)
(4)	<expr>	→	<var> <expr-tail>
(5)	<expr-tail>	→	− <expr>
(6)	<expr-tail>	→	ε
(7)	<var>	→	`ID` <var-tail>
(8)	<var-tail>	→	(<expr>)
(9)	<var-tail>	→	ε

(1)	S	→	$ABBA$
(2)	A	→	`a`
(3)		→	ε
(4)	B	→	`b`
(5)		→	ε

(1)	S	→	`a`Se
(2)		→	B
(3)	B	→	`b`B`e`
(4)		→	C
(5)	C	→	`c`C`e`
(6)		→	d

(1)	S	→	AB`c`
(2)	A	→	`a`
(3)		→	ε
(4)	B	→	`b`
(5)		→	ε

(1)	S	→	A`b`
(2)	A	→	`a`
(3)		→	B
(4)		→	ε
(5)	B	→	`b`
(6)		→	ε

Figure 5.8: Which grammars are LL(1)?

6.10 Exercises

6.10.1 LR(0)

Exercise 6.1. Build the CFSM corresponding to the following grammar:

(1)	S'	$\rightarrow$	$S\$$
(2)	S	$\rightarrow$	$\mathtt{a}C\mathtt{d}$
(3)		$\rightarrow$	$\mathtt{b}D$
(4)		$\rightarrow$	$C\mathtt{f}$
(5)	C	$\rightarrow$	$\mathtt{e}D$
(6)		$\rightarrow$	$F\mathtt{g}$
(7)		$\rightarrow$	CF
(8)	F	$\rightarrow$	$\mathtt{z}$
(9)	D	$\rightarrow$	$\mathtt{y}$

The algorithms to build a CFSM are found in Section 6.2.

Exercise 6.2. Give the action table of the LR(0) parser on the grammar of the previous exercise.

The building techniques for an LR(0) are in Section 6.3.

Exercise 6.3. Simulate the run of the LR(0) parser for the grammar of the previous exercises, on the word `aeyzzd`$.

See Section 6.3 again.

6.10.2 SLR(1)

Exercise 6.4. Build the SLR(1) parser for the following grammar (i.e., build the appropriate CFSM and give the SLR(1) action table):

(1)	S'	$\rightarrow$	$S\$$
(2)	S	$\rightarrow$	A
(3)	A	$\rightarrow$	$\mathtt{b}B$
(4)		$\rightarrow$	$\mathtt{a}$
(5)	B	$\rightarrow$	$\mathtt{c}C$
(6)		$\rightarrow$	$\mathtt{c}C\mathtt{e}$
(7)	C	$\rightarrow$	$\mathtt{d}A\mathtt{f}$

Is the above grammar LR(0)? Justify your answer.

SLR(1) parsers are covered in Section 6.4.

6.10.3 LR(k)

Exercise 6.5. Build the LR(1) parser for the following grammar (i.e., build the appropriate CFSM and give the LR(1) action table):

(1)	S'	$\rightarrow$	$S\$$
(2)	S	$\rightarrow$	$S\mathtt{a}S\mathtt{b}$
(3)		$\rightarrow$	$\mathtt{c}$
(4)		$\rightarrow$	ε

Is this grammar LR(0)? Is it SLR(1) 1? Justify your answers.

Section 6.5 is devoted to LR(k) parsers.

Exercise 6.6. Simulate the run of the parser you built at the previous exercise on the word `abacb`.

6.10.4 LALR(1)

Exercise 6.7. Build the LALR(1) parser for the grammar of exercise 6.5, using the LR(1) parser you have built for the same exercise.

The definition of LALR(1) parsers in Section 6.6 shows how to build them from LR(1) parsers.

Exercise 6.8. Find a grammar which is LR(1) but not LALR(1).

Since LALR(1) parsers can be built from LR(1) parsers, try to come up with states of an LR(1) parser that would be generate a conflict when the LALR(1) parser is built, and infer a grammar from that.