Universitetet i Oslo
Institutt for Informatikk

Reliable Systems
Martin Steffen

INF 5110: Compiler construction

Spring 2022

Series 1

2. 2. 2022

Topic: Scanning: automata and regular expressions (Exercises with hints for solution)

Issued: 2. 2. 2022

The exercises on this sheet refer to the lecture and material about *scanning* (or lexing). In general, when talking about letters, we assume for instance an English lettered alphabet, i.e., when speaking about "letters", we mean those letter characters as covered by the simple ASCII character set. Remember also: the ASCII character set is, technically, an *ordered* alphabet. Here, in this (and similar) exercises, we mostly restrict ourselves to alphabets of 2 or 3 different letters, only (like a, b, c).

Similarly, when talking about digits, we naturally refer to the numeric symbols of the base-10 numerical notation.

Exercise 1 (Regular expressions) Use *regular expressions* to capture the languages described informally as follows:

1. All strings of lowercase letters that begin and end with an a.
2. All strings of lowercase letters that begin or end with an a.
3. All strings of digits that do not contain leading zeroes.
4. All strings of digits that represents *even* numbers.
5. All strings of digits such that all the 2s precede all the 9s.
6. All strings of a's and b's (i.e., $\Sigma = \{a, b\}$) that don't contain 3 consecutive b's.
7. All strings of a's and b's that contain an odd number of a's *or* an odd number of b's.
8. All strings of a's and b's that contain an even number of a's *and* an even number of b's.
9. All strings of a's and b's that contain exactly as many a's than b's.

Solution: In general, most of the questions should be easy (but not all). At exam-time, this kind of questions should be routinely "under control" (and thus done quickly and without much hesitation). Some of the questions below are discussed in some length wrt. what the description actually means. In an exam, we try to avoid questions that requires too much head-scratching concerning how to actually interpret an English sentence.

As always for regular expressions (and actually for notational system of any kind), there's more than one way to express the same thing. If nothing specifically is said, one is allowed all "flavors" of regular expressions, including the more "user-friendly" ones. If we want some

specific form (for instance, the basic one), it will be stated explicitly. In the solution, I don't "bold-face" regular expressions (as done partly in the slides) to distinguish "syntax" (regular expressions) from "semantics" (words, languages). They are close to each other, anyway

1. *Letter a at the beginning and the end:*

$$a([a-z]^*a)? \ . \tag{1}$$

Be careful in that the regular expression consisting just of a single a should be allowed. I am halfway careful here also wrt. parentheses. Remember that the lecture specified "associativity" and "precedence" for the basic regular expressions. If one remembers those, one could have written $a[a-z]^*a$ because 1) $*$ binds strongest and 2) for concatenation, associativity does not play a role, as far as the resulting language is concerned. Since we did not speak about precedence of the option operator ?, the parentheses makes the grouping clear.

The notation $[\ldots]$ is *not* available in the basic regular expressions. If one needs a solution based on *basic* regular expressions, one could write

$$(a((a \mid b \mid c \mid \ldots \mid z)^*a)) \mid a \tag{2}$$

and also that is a "cheat" because obviously "..." is not allowed in regular expressions. In a written exam, it would be accepted as correct, if the ... are clear (as in this case). If one would write a lexer using a tool like `lex`, obviously the input syntax might *not* support "`...`". If one wanted to use there basic regular expressions, one has no choice to list them all! Fortunately, many tools will definitely support notations like `[a-z]`, if one has an *ordered* alphabet, that have the lower-case letters in subsequent slots in the encoding.[1]

2. *Letter a at the beginning* or *the end.*

$$(a[a-z]^*) \mid ([a-z]^*a) \tag{3}$$

In general, an "or" is easy and directly to translate (with |, of course).

If we allowed ourselves *named* regular expression (see the lecture), we could use that to give a name to $[a-z]^*$, which avoid writing it twice.

$$\begin{array}{l} \mathsf{lletters} = [a-z]^* \\ (a\ \mathsf{lletters}) \mid (\mathsf{lletters}\ a) \end{array} \tag{4}$$

Here we use an ad-hoc typographic convention, writing $\mathsf{lletters}$ to clearly distinguish it from *lletters* representing the sequence of individual lower-case "letters" l, l, e, etc.

3. *Strings of digits without leading zeroes.*

That's (almost) a standard one. What makes it actually unclear as exercise is what "leading 0" actually means in English. Such kind of unclarity is the reason why one is better off with writing ultimately regular expression to specify lexical conventions as opposed to prosa text. We interpret a "leading zero" as follow. A zero is leading if it occurs at the beginning of a string *and* is followed by more characters. A single 0 is *not* leading (but the word starts with an 0). Also, we wrote "string" which is a sequence of letters. We did not mention whether the string is allowed to be empty. It's plausible to say that the empty string has no "leading 0", therefore we include it as well. We can capture that as follows:

$$\epsilon \mid 0 \mid ([1-9]([0-9]^*)) \ . \tag{5}$$

[1] Food for thought: what about `[a-Z]`, assuming the standardized ASCII character set. We had shortly mentioned in the lecture.

Alternatively, we make use of named regular expressions, for instance

$$\begin{aligned} \mathsf{nonzero} &= 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ \mathsf{digit} &= 0 \mid \mathsf{nonzero} \\ \mathsf{noleadzero} &= \epsilon \mid 0 \mid \mathsf{nonzero}\ \mathsf{digit}^* \end{aligned} \tag{6}$$

Note: one classical definition of "number" would be "*non-empty* string of digits without leading zero". In contrast, our interpretation and solution included the empty string.

4. *All* even *numbers.*

   This time, it might be a bit unclear what "numbers" are. For instance, is 0000 a number? The concept of being "even" is more clear.

   The unclarity comes from the fact that actually, as "compiler writers" we are at this stage not so much dealing with *numbers* (as mathematical entities, which appears clear enough for most) but *notation* for numbers. When confronted with 00000, most mathematicians would say, "yes, that's a number" (actually they might say "it's 0", "it's zero", "it's null") and they might consider the discussion as outlandish, and anyway, nobody in a mathematical textbook would ever write 00000 anyhow, unless pressed.

   When fixing *notation* of numbers, the designer must make a choice: Is 00000 legal (= lexcially correct) for numbers or not? Should it be considered legal, at least then there is no debate about the fact that it's even. We take the standpoint here, that things like 00000 are *not* ok (i.e., the only number notation that can start with a zero is "0", something that most mathematical textbooks can relate to ...). With this assumption (which is not uncommon) we might get the following:

$$\begin{aligned} \mathsf{evendigit} &= 0 \mid 2 \mid 4 \mid 6 \mid 8 \\ \mathsf{nznumber} &= [1-9][0-9]^* \\ \mathsf{evennumber} &= (\mathsf{nznumber?}\ \mathsf{evendigit})\ . \end{aligned} \tag{7}$$

5. *All* 2*'s before all* 9*'s.*

   Also this one needs some careful consideration what is actually meant. Basically it's about the interpretation of what the English sentence is actually supposed to mean. So as not to worry: in an exam, we try hard to avoid such questions which are up-to speculation or else we accept all "reasonable" interpretations of such a sentence.

   The "ambiguity" seems to revolve around the formulation "all the 9's" or "all the 2's". Does that actually mean there need to be a 9 or 2?

   For instance: is the empty string allowed? If the sentence means:

   > if you ever see a 9, then it's forbidden to have a 2 afterwards.

   then the empty string seems ok. The following solution basically takes the previous sentence as specification for the task.

   There are 2 cases. If there are *no* 9's, it's ok. If there *is* a 9, afterwards no 2 is allowed.

$$\begin{aligned} \mathsf{digitnot9} &= 0 \mid 1 \mid \ldots \mid 8 \\ \mathsf{digitnot2} &= 0 \mid 1 \mid 3 \mid \ldots \mid 9 \\ \mathsf{solution} &= \mathsf{digitnot9}^* \mid \mathsf{digitnot9}^*\ 9\ \mathsf{digitnot2}^* \end{aligned} \tag{8}$$

   Other representations are possible, for instance

$$\mathsf{digitnot9}^*\ \mathsf{digitnot2}^*$$

6. *No 3 consecutive b's.* One possible representation is

$$(a \mid ba \mid bba)^*(\epsilon \mid b \mid bb) \tag{9}$$

The first part of the sequence is "end with an a, but don't do more than 2 b's". That is half-ok, as it requires that, if not empty, the word *has* to end with an a. That motivates the second part of the regular expression.

In such exercises, it pays off to specifically check the corner cases, like "is the empty string ok and is it captured by my solution expression?", "what if there is no bs at all?" etc.

7. *Odd number of a's or an odd number of b's.*

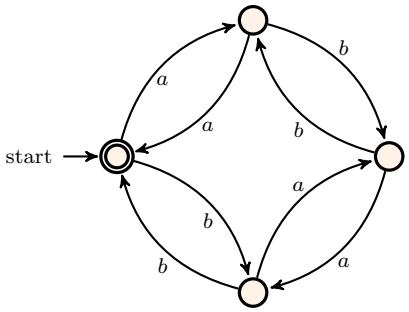
"Odd" means "one occurrence +2 more, and repeat that"

$$b^*ab^*(ab^*ab^*)^* \mid a^*ba^*(ba^*ba^*)^* \ . \tag{10}$$

8. Even number of a's *and* even number of b's.

This one is a bit more tricky. The source of the problem is that "and" is not easily translated into regular expressions. The emphasis is on "easily". It can be shown that regular languages are *closed* under intersection (which correspond to the word "and"). "Closed under intersection" means, given 2 regular languages, their intersection is regular. However, intersection is *not* part of the syntax of regular expression (not even to most "user-friendly" extensions). That means, one has to think by oneself (in this particular example) how one can capture the "and" requirement. There is a systematic algorithm to build for instance the automaton who represents the automaton capturing the intersection of its "sub-automata" What makes it trickier than the other constructs covered by regular expressions is that it's *not compositional*. Thus, one cannot cover intersection in the way that Thompson's construction translates regular expressions to NFA.

Perhaps the "easiest" way to get a solution is to go actually via an "automaton" (as opposed to try to nail down a regular expression directly): If one takes an "a", one cannot reach an accepting statet unless one takes another one (more precisely, an even number of those). And the same for b's. That may be the most straightforward route towards the following regular expression.



$$(aa \mid bb)^*(\ (ab \mid ba)\ (aa \mid bb)^* \quad (ab \mid ba)(aa \mid bb)^*\ )^* \ . \tag{11}$$

9. Same number of a's and b's.

That one cannot be represented by regular expressions (or FSAs). The lecture does not cover enough background to actually *prove* that this is not possible.[2] Intuitively, a FSA accepting the language would require that it can "count" and this would require that the automaton had unbounded memory.

[2]Impossibility arguments like that typically involve what is known as *pumping lemma* for regular languages. It's not part of this lecture and the pensum, though.

□

Exercise 2 (From regular expessions to automata) Assume the following regular expression

$$(\boldsymbol{a} \mid \boldsymbol{b})^* \boldsymbol{a} (\boldsymbol{a} \mid \boldsymbol{b} \mid \boldsymbol{\epsilon}) \tag{12}$$

Formulate a sentence (in English/Norwegian etc.) which captures the meaning of the regular expression. Now, turn the regular expression into the (deterministic, minimal) finite-state automaton which recognises that language. To do so, follow the 3 standard steps:

1. use *Thompson's construction* to obtain an non-deterministic finite-state automaton for the regular expression (12).
2. Use the subset construction to turn that into an equivalent DFA ("determinization").
3. Use the *partition refinement* algo to obtain the/a minimal equivalent DFA ("minimization")

Solution:

A first attempt might be

> All words on the two-letter alphabet a and b containing at least one a.

But actually that's not correct! One can wait until one sees the minimal automaton to see that it's unplausible

The first picture shows the outcome of Thompson's construction. Please note that the task is *not* simply to obtain a NFA "similar" to an automata expected from Thompson. The task is do give *the one* that comes out from the construction. Note how the construction leads to a quite an abundance of ϵ-transitions. Illustrative in this context is, for example, the part on the right-hand side of the overall automaton of Figure 1, corresponding to

$$\boldsymbol{a} \mid \boldsymbol{b} \mid \boldsymbol{\epsilon} \tag{13}$$

There are two remarks relevant in that context. The first concerns, as mentioned, the amount of ϵ-transitions. One transition, the one from state 18 to 19, is of course due to the fact that the regular expression mentions $\boldsymbol{\epsilon}$. The others are due to "glueing" the partially constructed automata for the subexpressions together. Note that the construction second one concerns the fact that the (sub-)automaton for equation (13) corresponds more precisely to the grouping

$$(\boldsymbol{a} \mid \boldsymbol{b}) \mid \boldsymbol{\epsilon} \tag{14}$$

where the parenthesis are introduced to make the structure more clear.[3] Since the situation is slightly ambigious, in a written exam, giving the automaton corresponding to $\boldsymbol{a} \mid (\boldsymbol{b} \mid \boldsymbol{\epsilon})$ instead of the grouping of equation (14) would be perfectly fine. However, a seemingless superfluous state like 12 would be required for a fully correct solution. A "naive" solution would probably leave out 12 and directly connect 11 with 13, 15, and 18.

The second step in the overall construction is *determinization* using the so-called *subset* or *powerset construction.* Due to the many ϵ-transitions, it's a bit cumbersome, and one has to be careful not to "forget" some reachable states, especially when dealing with the ϵ-closure. The result is shown in Figure 2. Missing is the indication, which are the final states (in follow-up figures, that is shown). The final or accepting states are all those that contain a final state from the original automaton (namely state 20). Unofficially: the numbers in the "meta-states" are meant as follows: the underlined ones are "directly" added by the constuion from the lecture (there written Q_a, the "a-successors for a set of states Q). The non-underlined ones are the ones that have to be added as ϵ-closure. See again the algorithm as presented in the lecture.

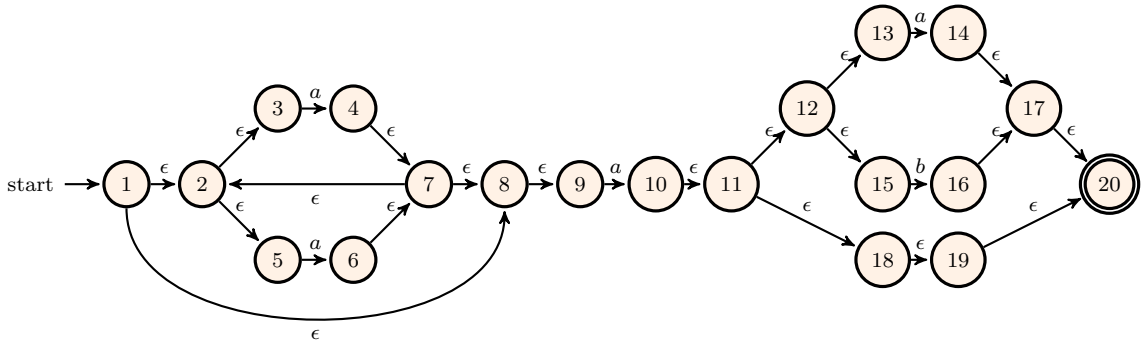


Figure 1: Result of Thompson's construction

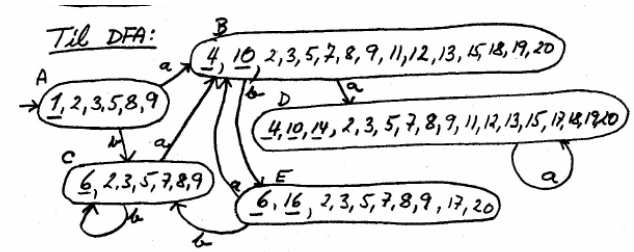


Figure 2: Result of determinization

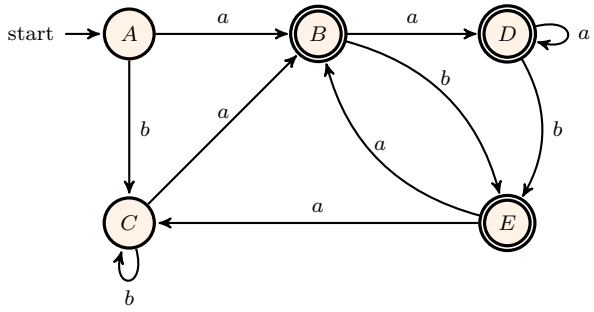


Figure 3: Result of the determinization

Finally, the *minimization.* The starting point is the deterministic one from the previous subtask. Of course, the "names" of the states are not very useful (the huge amount of numbers from the subset construction). So best is to rename the states. To avoid confusion, we better name them A, B etc. not 1, 2 ....

□

Exercise 3 Find a few "tools", or libraries for certain languages etc. that use regular expressions. Check the "syntax" of them.

Solution: Well, we don't provide a "solution" for that task. □

Exercise 4 (Reflection) The lecture and exercises teaches the standard way to turn regular expressions into deterministic and minimal finite-state automata, which goes through 3 separate

[3]Questions like whether parentheses belong officially to the syntax of regular expressions or whether they are used simply as convention to make the structure more explicit not does not concern us for now. However, similar questions will concern us in the context of parsing.

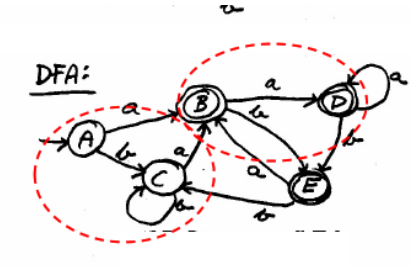


Figure 4: Minimization (1)

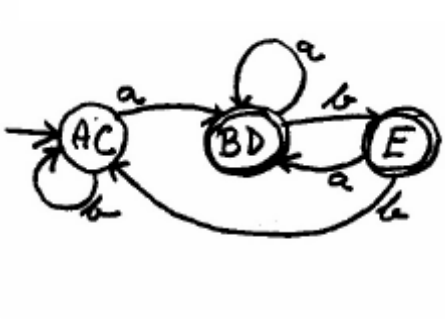
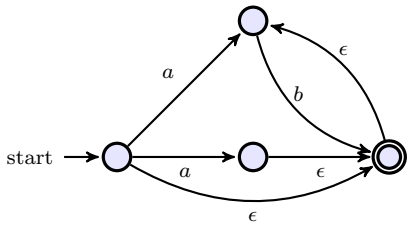


Figure 5: Minimization (2)

steps. A lexer (as first stage of most compilers) is typically interested only in the end-result, a minimal, deterministic FSA. Why do we bother (and why do many lexer/scanner implementations bother) to go through these 3 different stages, why not go directly from regular expressions to the end result?

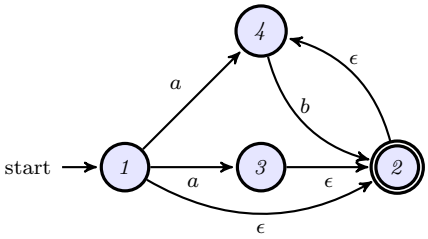
Solution: Also this has no "solution", it's more like thinking about it. □

Exercise 5 (Determinization) Determinize the automaton visualized in the following graphics. Use for that the powerset construction.

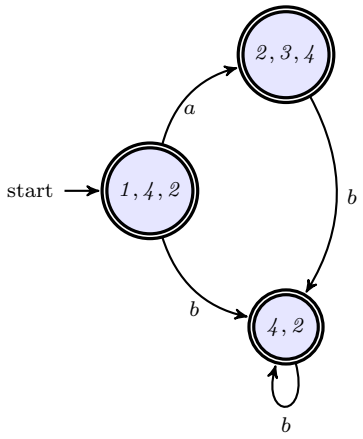


□

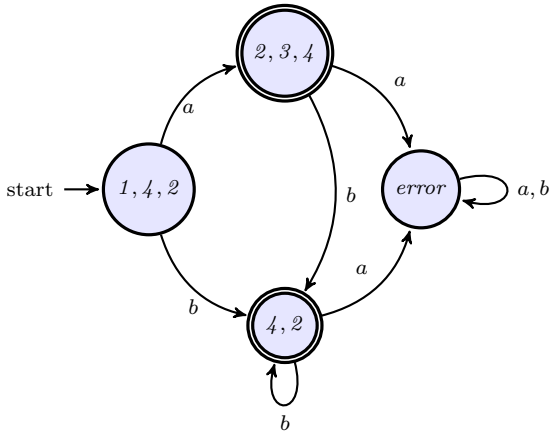
Solution: To start, we better make the states better identifiable by labelling them.



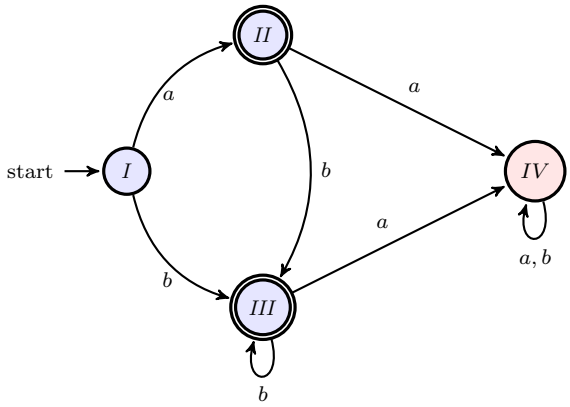
The next step is the power set construction. It requires in particular the notion of ϵ-closure. The resulting deterministic automaton looks as follows.



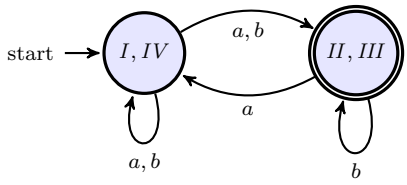
For minimization, then we need first *complete* the automaton. Only the 2 accepting states are not "complete" (insofar that transitions are missing). For completeness sake, also the additional error state on the right needs completion:



Minimization works via partition refinement, which is a simple iterative algorithm. It starts with a fixed, initial partitioning, and iteratively refines that until "stabilization". The initial partitioning is always the same. The states Q of the automaton are split (or partitioned) into 2 partitions: the final states on the one hand and the rest on the other. The states are now "labelled" by sets of numbers, and in the above automaton it gives the partitioning $\{\{2,3,4\},\{4,2\}\}$ on the one hand, and $\{\{1,4,2\},F\}$. We might also write and $\{q_{\{2,3,4\}}, q_{\{4,2\}}\}$ and $\{q_{\{1,4,2\}}, q_F\}$, drawing on the "convention" that states are often written as "q" and identified via a subscript. For a fresh start, we may relabel the states, using a fresh numbering again (this time using Roman numeral notation, not to confuse it with the original states). With the initial partitioning as before, namely $\{\{q_I, q_{IV}\},\ \{q_{II}, q_{III}\}\}$, we need to think, if that is already "stable" or if the algo need further step(s) splitting up the partitioning.

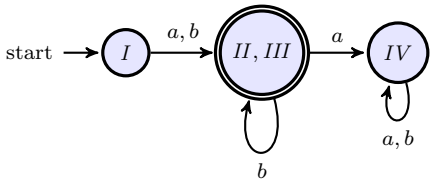


If we *collapsed* the states, i.e., if we'd put them together into two new states $q_{II,III}$ and $q_{I,IV}$, we'd get a 2-state automaton. We also need to have transitions, but they are just "inherited" from the non-collapsed one. That gives the following automaton:



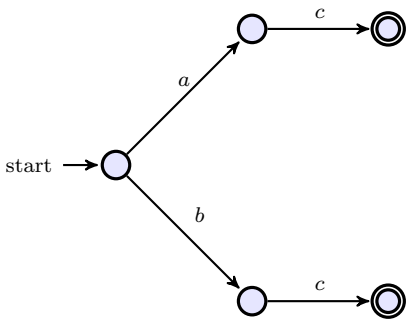
The problem with that automaton is, of course, that *it is not deterministic.* Also note: this automaton is *not* in general *equivalent* to the original one. Actually, it's not really "explicit" part of the minimization construction. It added here for *illustration* only.

But it illustrates where we have to split. State II, III is ok, the collapsed state I, IV is not. If we separate I from IV, we get a 3 state automaton. We cannot split I and IV further, since they are already consisting of single states. So therefore, that's the end of it



□

Exercise 6 (Minimization) Minimize the automaton vizualized in the following figure.



Solution: In the first step, we complete the automaton. Additionally, we add numbers to the states, for easier identification.

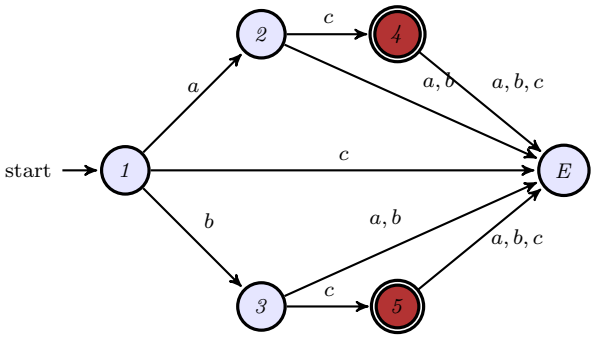


Figure 6: Completed automaton

This is the automaton, on which the partition refinement algorithm works. The first, initial partition splits the sets of states Q into *two* sets, the accepting states and the rest

$$\{1, 2, 3, E\} \quad \text{and} \quad \{4, 5\} \ .$$

The partitioning is indicated in the picture using two different colors. The starting point of the partition refinement algo is always the same, starting with a partitioning of 2 sets (final vs. non-final).

Now, the partitioning will be iteratively "refined" (i.e., existing partitions will be further split) if need be until no further split is necessary ("stabilization"). At that point, the minimal deterministic automaton will have been reached. The conditions on splitting have been discussed in the lecture. Intuitively, one could see it as follows. Taking the initial partitioning into account, one can consider, if that would give rise to a minimal deterministic automaton already. The corresponding 2-state automaton is shown in Figure 7. It's clear that the automaton from that figure cannot be the minimal deterministic automaton, simply because it's not deterministic! The left state has two different c-successors. That's an indication, that the state $\{1, 2, 3, E\}$ has to be split.

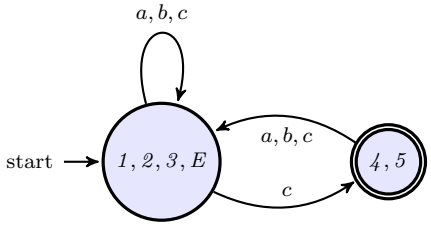


Figure 7: Initial partitioning

It should be noted that the collapsed automaton from Figure 7 is shown more for illustration that motivates the splitting condition. The automaton illustrates *that* the state $\{1, 2, 3, E\}$ needs to be split and even that the cause of the split is due to transition labelled c, but it not does not contain enough information to determine *how* exactly it needs to be split. We need not consult the more detailed Figure 6 for that.

Considering the behavior of the "elements" state $\{1, 2, 3, E\}$ wrt. c (i.e., the states 1, 2, 3, and E from Figure 6), we see that one needs to split it further ("refine") into $\{1, E\}$ and $\{2, 3\}$. Using a collapsed representation again for illustration, this yields the automaton of Figure 8.

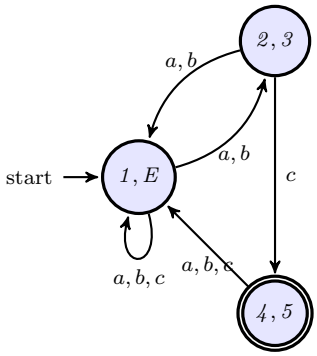


Figure 8: After one refinement step

Again it's clear that it cannot be the end of it, state $\{1, E\}$ behaves non-deterministically wrt. a resp. b. Now we can split with resp. to a (for example), which gives the automaton from the next picture 9

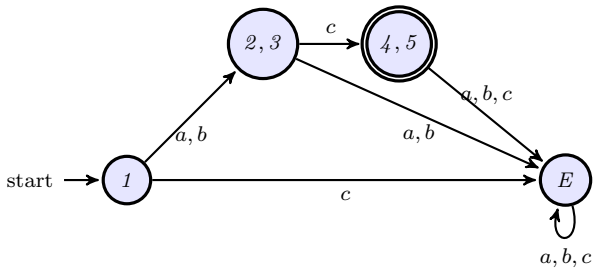


Figure 9: Stable

It turns out that doing that split wrt. a solved also the non-determinism wrt. b, and the results is a deterministic automaton, no need for further split. □