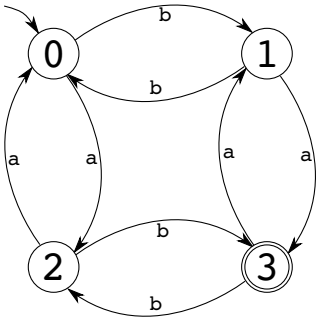
Assignment 1

Compiler Design I (Kompilatorteknik I) 2011

1 Automata

Give a finite state automaton (deterministic or non-deterministic) that accepts all the strings consisting of the symbols 'a' and 'b' that have an odd number of 'a's and an even number of 'b's.

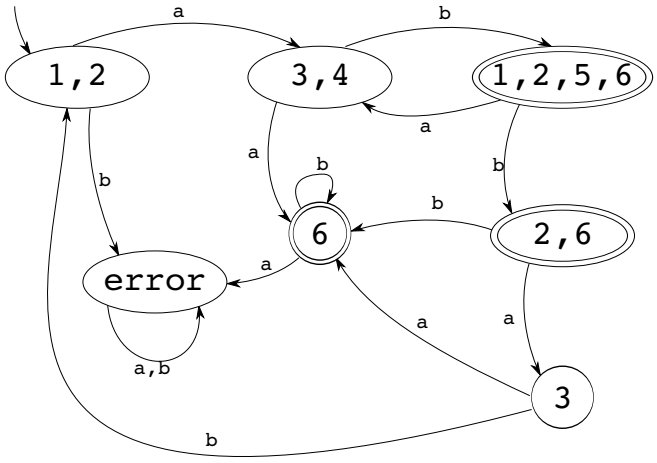
1.1 Solution

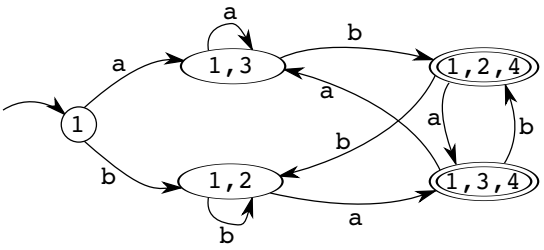


2 NFA to DFA conversions

Convert the following NFAs to the equivalent DFAs.

2.1 Solution





3 Regular Expressions

The characters '.' and '+' have special meaning in regular expressions, as explained in the lesson. When we want to refer to the characters themselves in a regular expression we prepend them with a '\'. A regular expression that matches all strings that have a literal '.' followed immediately by a literal '+' is for example: ".*\.\+.*"

With that in mind, give a regular expression that describes all e-mail addresses that comply with the following rules:

- they can be written in the form X@Y
- the X part consists only of alphanumeric characters, dots and the character '+'
- the Y part consists only of alphanumeric characters and dots
- a dot or '+' can appear only between two alphanumeric characters
- at most one '+' appears in X
- at least one dot appears in Y

Examples:

```
Valid:          | Invalid:
  foo@ba.r      |   foo
  f.o.o@b.a.r   |   f!oo@b^ar
  f+oo@b.ar     |   foo@bar+y
                |   f..oo@b.a
                |   foo+@b.a
                |   f+o+o@b.ar
                |   foo@bar
```

You can use all the other special characters presented in the lectures and the lesson. Keep in mind that you can combine ranges (as they appear in Lecture 3, slide 3): [a-z]|[A-Z] == [a-zA-Z]

3.1 Solution

```
c = ([a-zA-Z0-9]+)

dots     = c(\.c)*
min1dots = c(\.c)+
mst1plus = c(\+c)?


X:
```

```
d = c(\.c)*
X = d(\+d)?

X = d(\+d)?
X = (c(\.c)*)(\+(c(\.c)*))?
X = ([a-zA-Z0-9]+(\.[a-zA-Z0-9]+)*)(\+([a-zA-Z0-9]+(\.[a-zA-Z0-9]+)*))?

Y:
X = c(\.c)+
X = [a-zA-Z0-9]+(\.[a-zA-Z0-9]+)+

regex = X@Y
regex =
([a-zA-Z0-9]+(\.[a-zA-Z0-9]+)*)(\+([a-zA-Z0-9]+(\.[a-zA-Z0-9]+)*))?
@
[a-zA-Z0-9]+(\.[a-zA-Z0-9]+)+
```

4 Understanding DFAs

Standard DFAs must have exactly one transition out of each state for each input symbol. Suppose we were to revise this definition to allow zero or one transition out of each state on each input symbol. Some regular expressions would then have smaller "DFAs" than they do under the standard definition. Give an example of one such regular expression.

5 Lexical analysis

In the example we saw in the lesson, we tried to encode information about the tokens that are emitted by the lexical analyzer as outputs for the DFA. In practice, the algorithm is somewhat different: The DFA of the lexical analyzer does not contain extra output information, just accepting states that have an added label about what category of tokens will be emitted if the input ends at these states. Using this DFA we begin to read the input, continuing for as long as we don't fall into the null state. If the next input symbol would lead to a transition in the null state, we instead back up through the sequence of states we entered and, as soon as we find an accepting DFA state, we output a token of the category in its label and start again with the rest of the input (adding of course any symbols we "took back" while backing up). An example is shown in section 3.8.3 of Aho, Lam, Sethi, Ullman *Compilers Pricnciples, Techniques & Tools*.

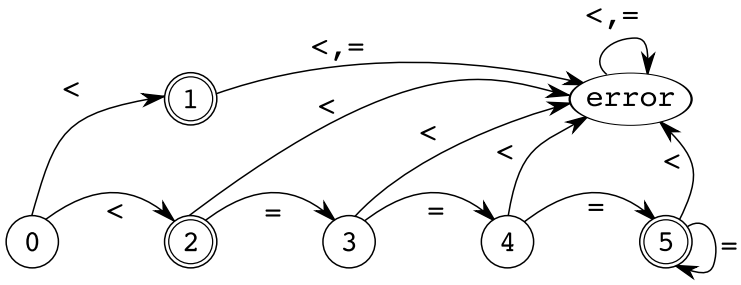
With these in mind, let L be the language of strings on $\Sigma = \{<, =\}$ defined by $L = \{<, =, <====^*\}$, that is $\{<, =\} \cup \{<=^n \mid n \geq 3\}$.

1. Construct a DFA that accepts L. *(Hint: Begin with an NFA and then convert it)*
2. Describe how the lexical analyzer will tokenize the following inputs:
   - <====
   - ==<==<==<==<==
   - <=====<

5.1 Solution

Important: The language L does only contain every token once and not arbitrary often as stated in the lesson. A fitting DFA is the following:

DFA



Tokenized Input

- <==== - The whole sequence is accepted as one token. (no backtracking)
- =, =, <, =, =, <, =, =, <, =, =, <, =, = - The lexer reads = and will output this token, since every other input would lead into the error/null state. This happens again for the next input character. The lexer would then read the next three characters of the input. Since the next input character would lead into an error/null state, the lexer backtracks to the last accepting state an outputs the corresponding token <. This happens 4 times.
- <=====, < The lexer reads the first 6 input characters and encounters a transition into the error/null state for the next input character. The token <===== is found by backtracking and will be outputted. The last < character is accepted by the lexer and outputted.