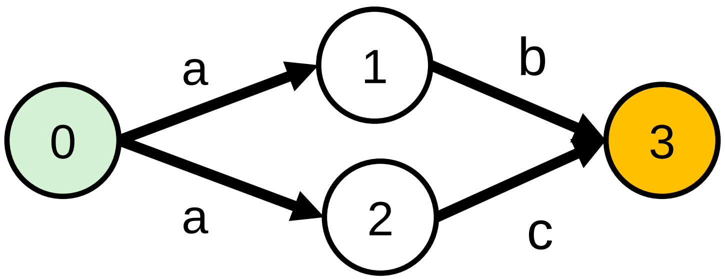
More on Regular Expressions

Nondeterministic?

- Why are NFAs called *nondeterministic* finite automata?
- Because our definition of NFA allows states to have multiple outgoing transitions with the same label:



- This makes it easier to construct NFAs for RE like ab | ac
- But how to implement that? For the input “ab”, should we go to state 1 or state 2 after reading the first character “a”?
- Another example: a* a

- *Kleene’s theorem*, part 2: Every NFA can be transformed into a Deterministic FA (DFA; a FA without nondeterministic transitions)

Transforming NFAs into DFAs

- Idea: For a state s_1 with transitions

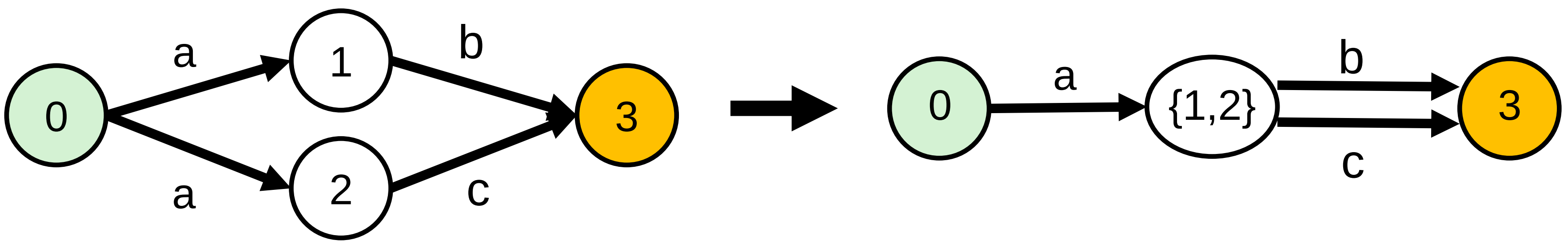
$$s_1 \xrightarrow{\text{a}} s_2$$

$$s_1 \xrightarrow{\text{a}} s_3$$

we create a new state $\{s_2, s_3\}$ that represents the fact that we can be in s_2 or in s_3 after receiving “a”.

This construction of state-combinations is called *powerset construction.*

- Example: NFA for RE ab|ac



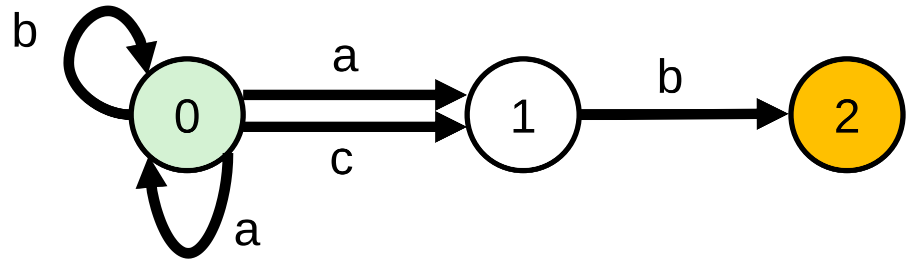
Powerset Construction

- For an NFA without ε-transitions with
  - Finite set S of states
  - Initial state $s_0 \in S$
  - Set $F \subseteq S$ of final states
  - Input alphabet Ω
  - Set of transitions $T: S \times \Omega \times S$

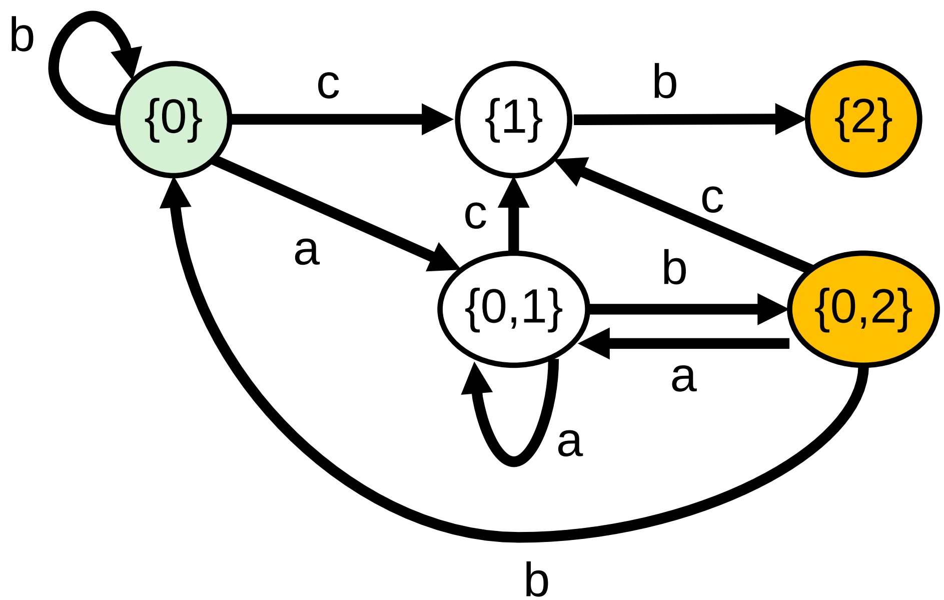
  its *powerset automaton* on the same alphabet Ω is given by
  - Finite set $S' = 2^S$ of states
  - Initial state $\{s_0\}$
  - Final states $F' = \{Q \subseteq S \mid Q \cap F \neq \emptyset\}$
  - Set of transitions T' with:
    - $\forall Q \subseteq S, c \in \Omega$: $(Q, c, R) \in T'$ for $R = \{t \mid s \in Q, (s, c, t) \in T\}$

Example 2: Powerset construction

NFA



DFA



Complexity of the Powerset Construction Method

- Complexity of construction of DFA for a RE:
  1. Construct the NFA for the RE:
     time and space $O(length\ of\ RE)$
  2. Powerset construction:
     time and space $O\left(2^{length\ of\ RE}\right) = O(2^n)$
     where n is the number of states of the NFA

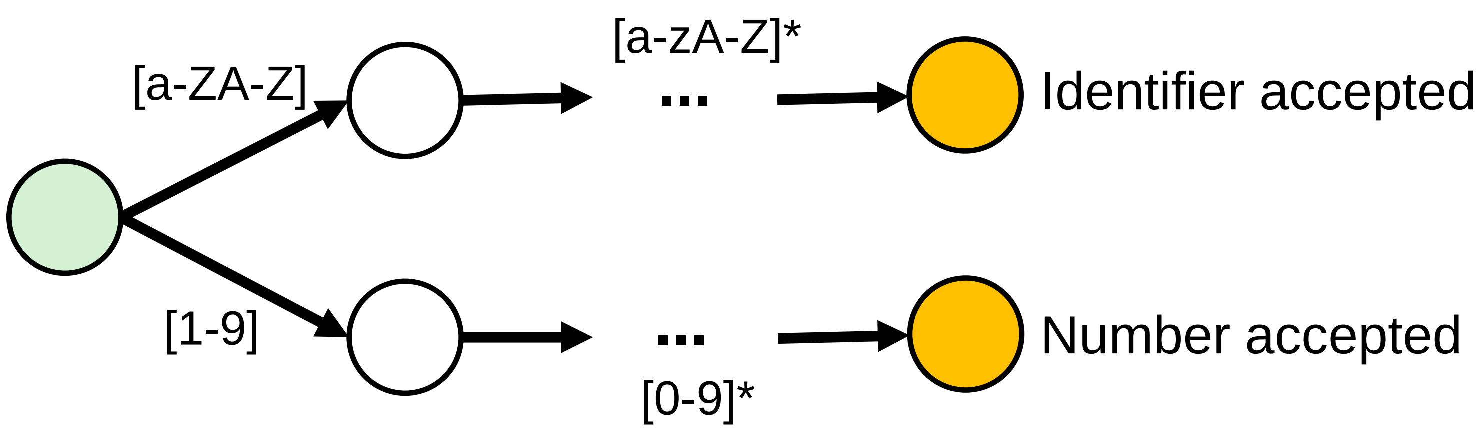
- Complexity at runtime, when checking whether the DFA accepts an input
  - Time complexity: $O(length\ of\ input)$
  - Space complexity:
    - $O(2^n)$ to store the DFA
    - $O(1)$ to remember the current state of the DFA

An Alternative Approach

- For complex RE, the DFA can become very large. Exponential time and space complexity! (although rarely, in practice)
- Alternative approach:
  - Do not create the DFA in advance
  - When running an input $a_1 a_2 a_3 \ldots a_n$ through the NFA, keep track of the possible states we can be in, e.g., {1,2}
  - Basically, this means we are constructing the powerset during runtime
- Advantage: no DFA construction needed, only $O(length\ of\ RE)$ for NFA construction
- Disadvantage: more bookkeeping during input processing

Practical aspects

- For a lexer, we can use multiple final states to handle symbol classes
- Example: [a-zA-Z] [a-zA-Z]* | [1-9] [0-9]*
  - Identifier: [a-zA-Z] [a-zA-Z]*
  - Number: [1-9] [0-9]*



Extended Matching Problem

- For a lexer, we usually have several RE, e.g.,

  ForKeyword: “for”
  Identifier: [a-zA-Z] [a-zA-Z]*
  Number: [1-9] [0-9]*
  WS: " "

  We want to decompose the input by repeatedly applying the RE
- Example:
  - Input: for 2472 ab
  - Desired decomposition:

    <ForKeyword,> <WS,> <Number,2472> <WS,> <Identifier,ab>
- Observation: The decomposition is not unique
  - <Identifier,“for”> <WS,> <Number,247> <Number,2> <WS,> ...

Making the decomposition unique

- We apply the principle of *First Longest Match*
- Example:

  ForKeyword: “for”
  Identifier: [a-zA-Z] [a-zA-Z]*
  Number: [1-9] [0-9]*

  The possible lexems are given by:

  “for” | [a-zA-Z] [a-zA-Z]* | [1-9] [0-9]*

- Approach:
  - We choose the longest match possible

    2472 will be lexed to <Number,2472> (and not 247 and 2)
  - We choose the first matching option (from left to right)

    <u>for</u> will be lexed to <ForKeyword,> and not to <Identifier,“for”>

Longest match not always adequate in some languages

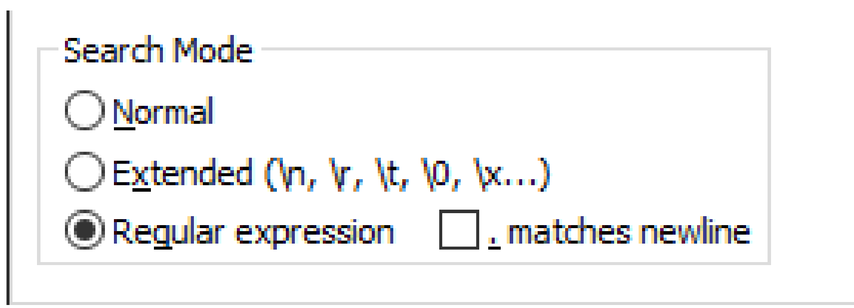
- Example:

x = y/*z	Begin of comment or y divided by *z ?
List<List<Integer>>	Two ">" or right-shift-operator ">>" ?

- Requires special care in the lexer
- Annoying, better avoid this when designing a language

Practical aspects, part 2

- In this course, you will implement a lexer by hand, but the automatic translation of arbitrary REs to NFAs or DFAs is very useful in practice
- Applications:
  - Many compiler authors use a *lexer generator* tool (ANTLR, flex,...) to automatically translate REs to lexer code
    - Such tools also minizime the DFA, i.e., remove duplicate states (not discussed here)
  - Programs that handle user-defined REs. Examples: the search function in text editors, the Pattern class in the JDK,...



Implementation by hand

```
Symbol getNextSymbol() {
        char c = r.readChar();
        if(c>='1' && c<='9') {
                String s = "";
                while(true) {
                        s = s + c;
                        c = r.readChar();
                        if(c<‘0’ || c>‘9’) {
                            r.unread(c);
                            break;
                        }
                }
                return new Symbol(Number, Integer.parseInt(s))
        }
        else if((c>=‘a' && c<=‘z’) || (c>=‘A' && c<=‘Z’)) {
                ...read the rest of the identifier...
                return new Symbol(Identifier,...)
        }
        else ...
```

Fortunately, for many source languages, looking at one character is enough to decide which Symbol class we have

Longest match. We read as much as we can

This requires a PushbackReader

Requires some special treatment for keywords here