Lexing

Goal

- Input: source code as a character sequence

  a = 3 + sin(mx)

- Goal: split source code into lexical structures (*lexemes*)

  a , = , 3 , + , sin , (, mx ,)

- Output: a sequence of syntactic atoms (*symbols*) represented by those lexemes
  - Often implemented as: Symbol = <Token, Attribute>
- For our example, we get this sequence of symbols:

  <Identifier, “a”>
  <AssignmentOperator,>
  <Number, 3>
  <ArithmeticOperator, “+”> (or just <AddOperator,>)
  <Identifier, “sin”>
  <SpecialCharacter, “(“> (or just <OpenParenthesis,>)
  <Identifier, “mx”>
  <SpecialCharacter, “)”> (or just <CloseParenthesis,>)

Lexical Analysis

- There are compilers without a separate parser and lexer
  - The parser could directly read and process the source code
- In practice, it can be useful to have a lexer
  - Simplifies the job of the parser. The lexer can do a first preprocessing of the source code, for example filter out whitespace characters, newlines, comments,...
  - The lexer is less complex than the parser and can be implemented very efficiently
- Note. You don’t lex the entire source code before starting the parser.
- Typical implementation:

```
class Lexer {
    Lexer(Reader sourcefile) { ... }
    Symbol getNextSymbol() { ... }
}
```

Defining lexemes

- We can use a *regular expression* to describe the allowed lexemes in the source code

“=“ | “+” | “-” | “++” | “(“ | “)” | [a-zA-Z] [a-zA-Z]* | ...

Choice

Short for "+" "+"

Short for: “a” | “b” | ...

Zero or more repetitions (Kleene star)

- For a lexer, it is usually more convenient to write one RE per symbol class:

AssignmentOperator:	"="
IncrementOperator:	"++"
ArithmeticOperator:	"+" | "-"
SpecialCharacter:	"(" | ")"
Identifier:	[a-zA-Z] [a-zA-Z]*
Number:	[1-9] [0-9]*
Whitespace:	(“ ” | “\t” | “\n”)*

Formal Definition of Regular Expressions (RE)

- Given: an alphabet Ω of allowed characters $a, b, \ldots \in \Omega$
- The set of regular expressions RE over Ω is defined as:
  - $\varepsilon \in RE$ (empty expression)
  - $\Omega \subseteq RE$
  - If $\alpha, \beta \in RE$ then also
    - $\alpha\ \beta \in RE$
    - $\alpha \mid \beta \in RE$
    - $\alpha^* \in RE$

Regular languages

- A regular expression specifies a *regular language*, i.e., the possible sequences of characters described by that regular expression.
- The mapping $[\![.]\!]$: $RE \rightarrow 2^{\Omega^*}$ gives the *semantics* of a regular expression
  - $[\![\varepsilon]\!] = \{\}$ (empty language)
  - $[\![a]\!] = \{\mathrm{a}\}$
  - $[\![\alpha\ \beta]\!] = [\![\alpha]\!]\ [\![\beta]\!]$
  - $[\![\alpha|\beta]\!] = [\![\alpha]\!] \cup [\![\beta]\!]$
  - $[\![\alpha^*]\!] = [\![\alpha]\!]^*$
- Example: The RE

  [1-9] [0-9]*

  specifies the regular language

  $[\![$[1–9] [0–9]$*]\!]$ = { 1, 2, 3, ..., 9, 10, 11, 12, ..., 1340242, 4595983,... }

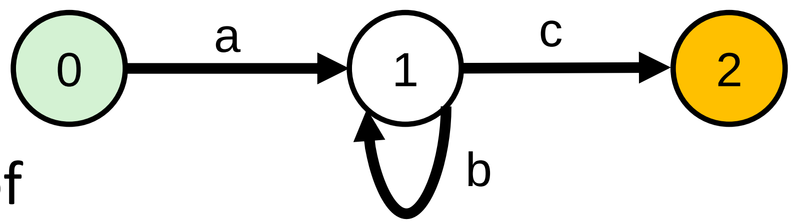
Simple Matching Problem

- Before we show how we can implement a lexer using REs, let's look at a simpler problem:
  - *I give you a sequence of characters and a RE and you have to decide whether the sequence is in the language of the RE*
  - Often used to validate user input, for example a phone number
- Example: Input: 1233072, RE: [1-9] [0-9]*
- Implementation:

```
boolean match(Reader r) {
    char c = r.read();  // read returns -1 if end of input reached
    if(c<'1' || c>'9') return false;
    while(true) {
        c = r.read();
        if(c==-1) break; // end of input reached
        if(c<'0' || c>'9') return false;
    }
    return true; // all good
}
```

Nondeterministic Finite Automaton (NFA)

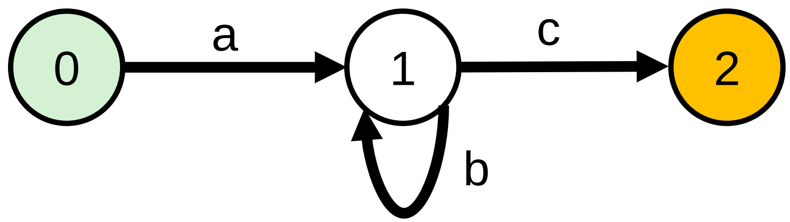
- Can any RE be translated to simple code? A more theoretical approach is needed for this
- Here is an example Nondeterministic Finite Automaton (NFA)



- An NFA consists of
  - 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: $S \times \Omega \cup \{\varepsilon\} \times S$
- If the NFA is in state $s \in S$ and consumes a character c it moves to state $t \in S$ provided there is a transition $(s, c, t) \in T$

  We write $s \xrightarrow{c} t$

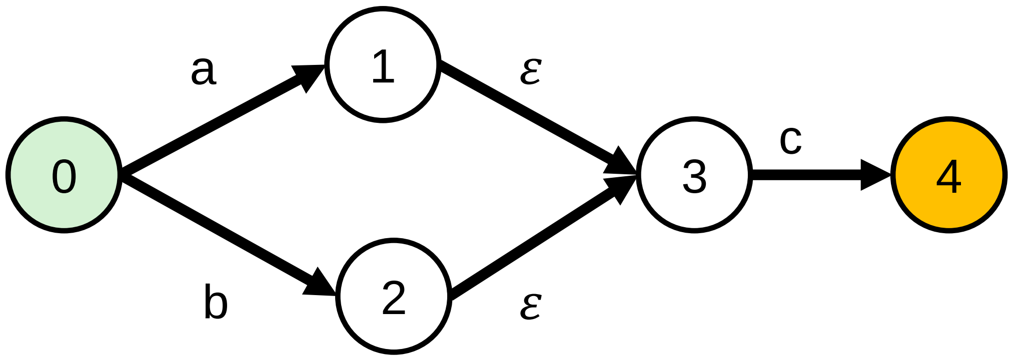
Simple Matching Problem with NFA



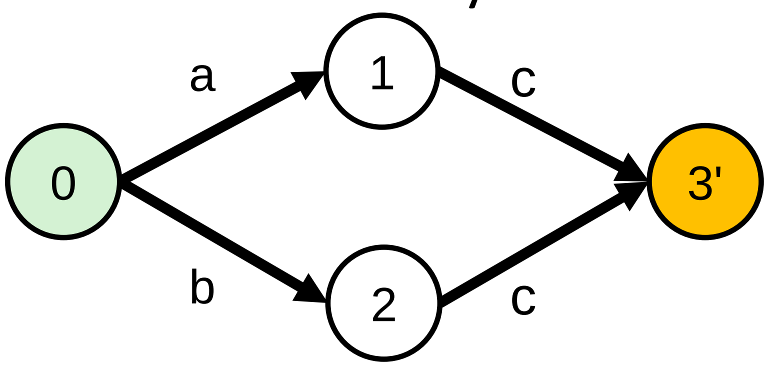
- For the input $a_1a_2a_3 \dots a_n$ with $a_i \in \text{alphabet } \Omega$, we say that the NFA accepts $a_1a_2a_3 \dots a_n$ if there is a sequence of transitions from the initial state to a final state that consume $a_1a_2a_3 \dots a_n$
- Example: Input abbbc
  - Sequence of transitions: $0 \overset{\text{a}}{\rightarrow} 1 \overset{\text{b}}{\rightarrow} 1 \overset{\text{b}}{\rightarrow} 1 \overset{\text{b}}{\rightarrow} 1 \overset{c}{\rightarrow} 2$ Accepted!
- Example: Input abbba
  - Sequence of transitions: $0 \overset{\text{a}}{\rightarrow} 1 \overset{\text{b}}{\rightarrow} 1 \overset{\text{b}}{\rightarrow} 1 \overset{\text{b}}{\rightarrow} 1 \overset{a}{\rightarrow}$ ✕ Not accepted!
- Congratulations! We have built an NFA that accepts the language of the RE a b* c
- *Kleene's theorem*: the set of language recognized by NFAs is identical to the set of languages of REs, i.e., we can always construct an NFA for an RE

Choice

- A RE of the form a | b can be represented by an NFA with multiple transitions leaving a state and ε-transitions
- ε-transitions are "empty' transition. They don't consume characters.
- Example: NFA with five states for the RE (a|b) c

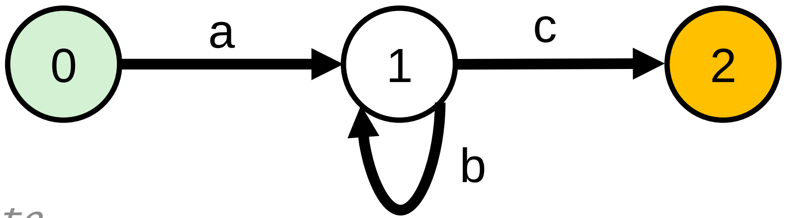


- Note that ε-transitions can be always removed:



Implementing the Simple Matching Problem

- Thanks to the representation of the RE by a Finite Automaton, we can easily write down an implementation (or even write a tool that automatically translates a RE to code):




```
boolean simpleMatch(Reader r) {
    int state = 0; // initial state
    while(true) {
        char c = r.read();
        if(c==-1) break;    // end of input reached
        if(state==0) {
            if(c=='a') state = 1;
            else return false;// input not accepted
        }
        else if(state==1) {
             if(c=='b') state = 1;
             else if(c=='c') state = 2;
             else return false;// input not accepted
        }
        else return false;
    }
    return state==2; // in final state?
}
```