CSE216
Programming Abstraction

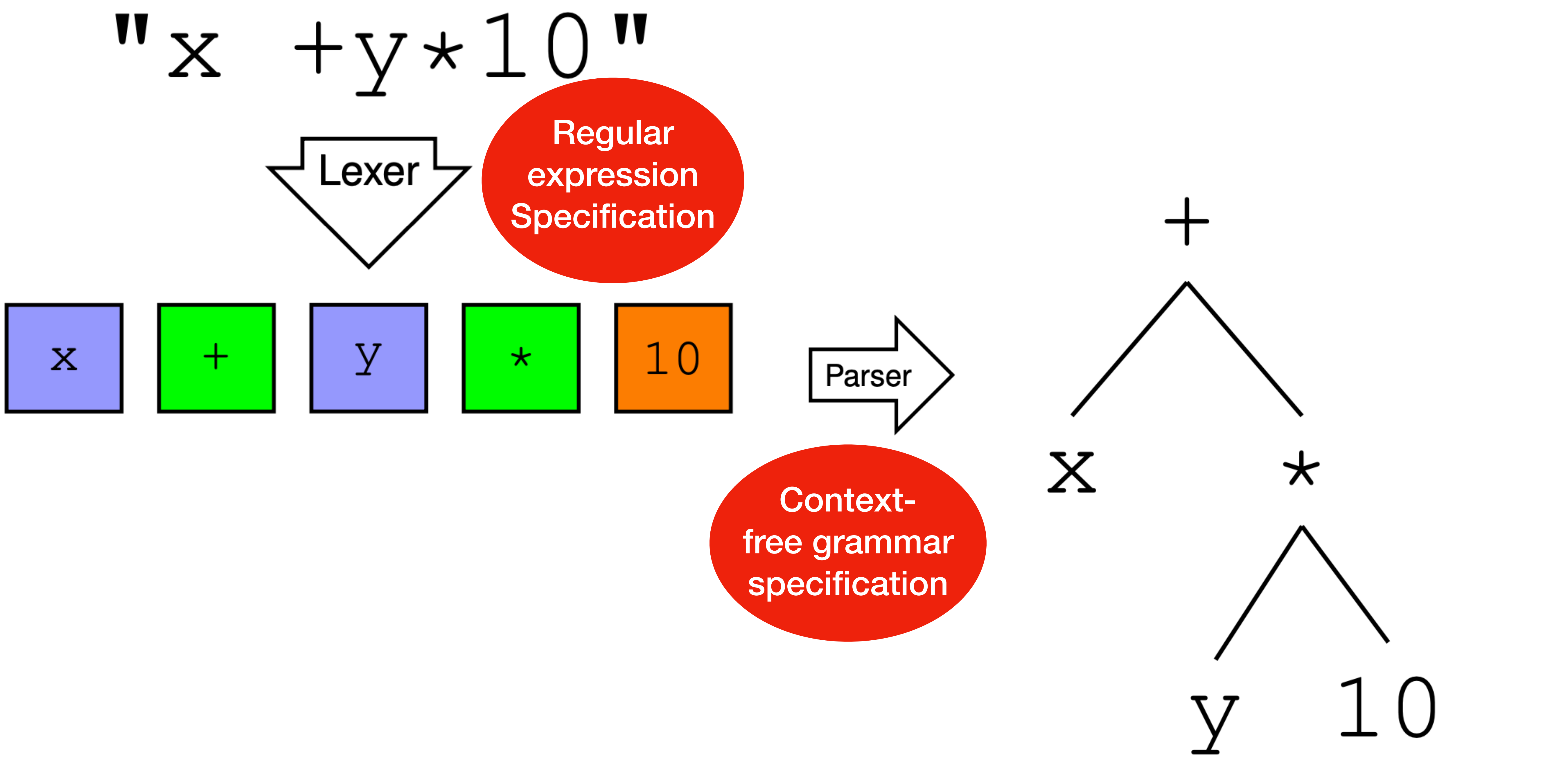
Instructor: Zhoulai Fu

State University of New York, Korea

What is a program?

- Let us consider an expression x + y*10
- Think of this expression as a program in a programming language
- This is actually a program written in a programming language used by a calculator
- Today we will analyze the syntax of a general program — Syntax analysis
- Syntax analysis can take a whole semester to learn; we will touch only the surface

Syntax analysis



Regular expression specification looks like this in Ocaml

```
rule Token = parse
  | [' ' '\t' '\n' '\r']  { Token lexbuf }
  | ['0'-'9']+            { CSTINT (...) }
  | ['a'-'z''A'-'Z']['a'-'z''A'-'Z''0'-'9']*
                          { keyword (...) }
  | '+'                   { PLUS  }
  | '-'                   { MINUS }
  | '*'                   { TIMES }
  | '('                   { LPAR  }
  | ')'                   { RPAR  }
  | eof                   { EOF   }
  | _                     { lexerError lexbuf "Bad char" }
```

Context-free grammar specification looks like this in Ocaml

```
Main ::= Expr EOF
Expr ::= NAME
       | CSTINT
       | - CSTINT
       | ( Expr )
       | let NAME = Expr in Expr end
       | Expr *  Expr
       | Expr + Expr
       | Expr - Expr
```

Menu for Today

- Regular expressions
- Finite State Automata
- Nondeterministic Finite Automaton (NFA)
- Deterministic Finite Automaton (DFA)

- Python basics

Regular expressions

r	Meaning	Language $\mathcal{L}(r)$
$\texttt{a}$	Character $\texttt{a}$	$\{"\texttt{a}"\}$
ε	Empty string	$\{""\}$
$r_1 r_2$	r_1 followed by r_2	$\{s_1 s_2 \mid s_1 \in \mathcal{L}(r_1), s_2 \in \mathcal{L}(r_2)\}$
r^*	Zero or more r	$\{s_1 \ldots s_n \mid s_i \in \mathcal{L}(r), n \geq 0\}$
$r_1 \vert r_2$	Either r_1 or r_2	$\mathcal{L}(r_1) \cup \mathcal{L}(r_2)$

Examples

$\texttt{ab}^*$ represents $\{"\texttt{a}", "\texttt{ab}", "\texttt{abb}", \ldots\}$

$(\texttt{ab})^*$ represents $\{"", "\texttt{ab}", "\texttt{abab}", \ldots\}$

$(\texttt{a}|\texttt{b})^*$ represents $\{"", "\texttt{a}", "\texttt{b}", "\texttt{aa}", "\texttt{ab}", "\texttt{ba}", \ldots\}$

Exercise

What does $(\texttt{a}|\texttt{b})\texttt{c}^*$ represent?

Regular expression abbreviations

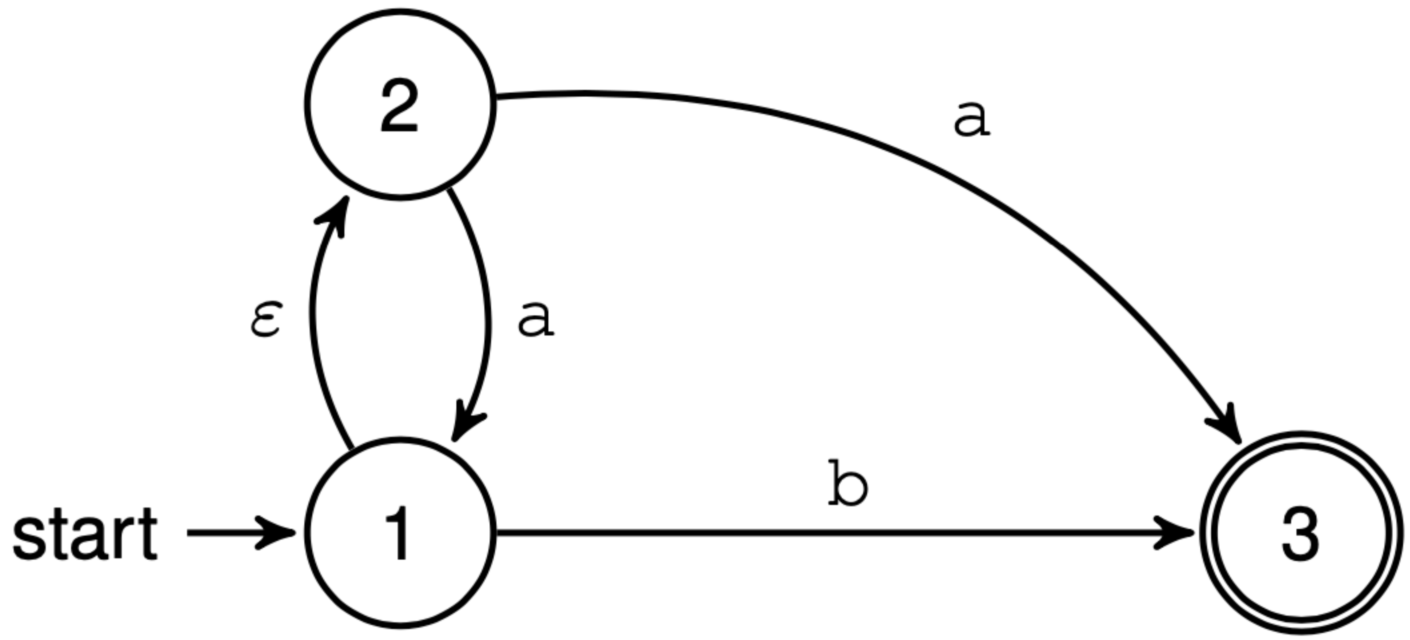
Abbrev.	Meaning	Expansion
`[aeiuo]`	Set	$\texttt{a}\vert\texttt{e}\vert\texttt{i}\vert\texttt{o}\vert\texttt{u}$
`[0–9]`	Range	$\texttt{0}\vert\texttt{1}\vert\ldots\vert\texttt{8}\vert\texttt{9}$
`[0–9a–Z]`	Ranges	$\texttt{0}\vert\texttt{1}\vert\ldots\vert\texttt{8}\vert\texttt{9}\vert\texttt{a}\vert\texttt{b}\vert\ldots\vert\texttt{y}\vert\texttt{z}$
$r?$	Zero or one r	$r\vert\varepsilon$
r^+	One or more r	$r\,r^*$

Exercises

Write regular expressions for:

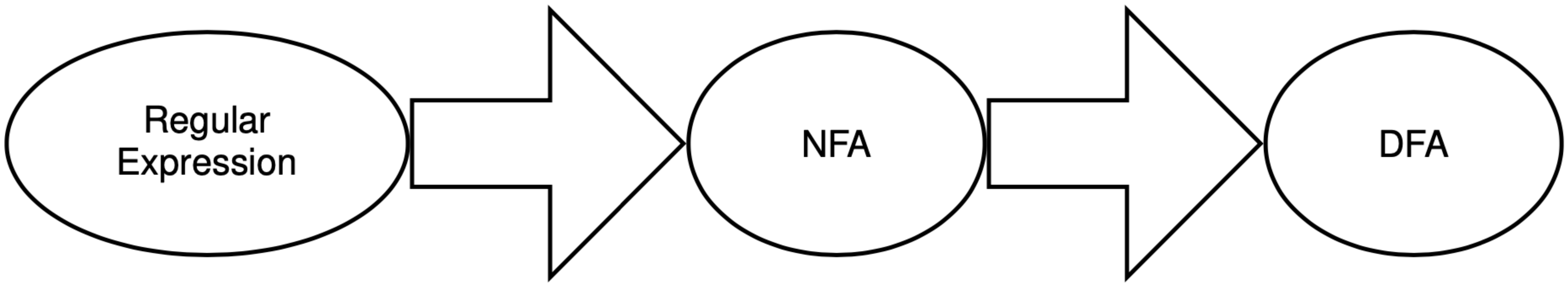
- Non-negative integer constants
- Integer constants
- Floating-point constants:
  - `3.14`
  - `3E8`
  - `+6.02E23`
- Java variable names:
  - `xy`
  - `x12`
  - `_x`
  - `$x12`

Finite State Automata



- A finite automaton, FA, is a graph of states (nodes) and labelled transitions (edges)
- An FA accepts string s if there is a path from start to an accept state such that the labels make up s
- Epsilon (ε) does not contribute to the string
- This automaton is nondeterministic (NFA)
- It accepts string b
- Does it accept a or aa or ab or aba?

Regular expression = finite automata



- For every regular expression r, there exists a deterministic finite automaton that recognizes precisely the strings described by r.

- The converse is also true.

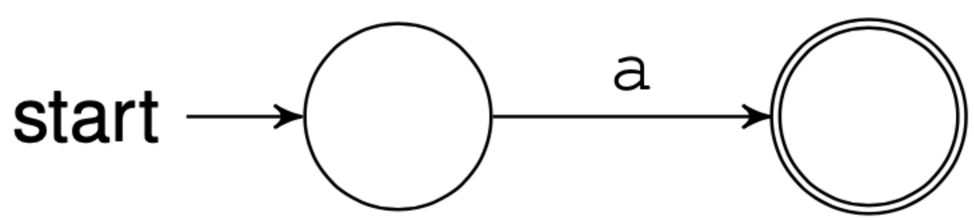
- Construction: Regular expression ⇒ Nondeterministic finite automaton (NFA) ⇒ Deterministic finite automaton (DFA)

- Results in an efficient way of determining whether a given string is described by a regular expression

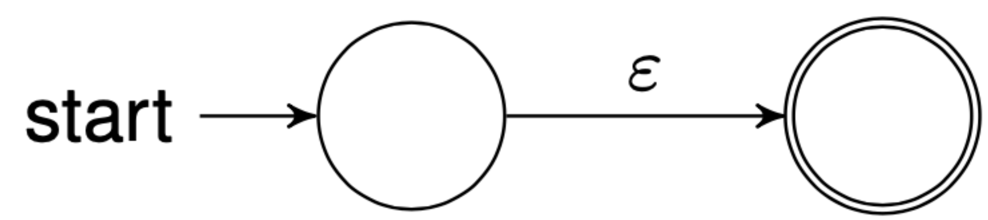
From regular expression to NFA

Build NFA recursively by the case of the regular expression.

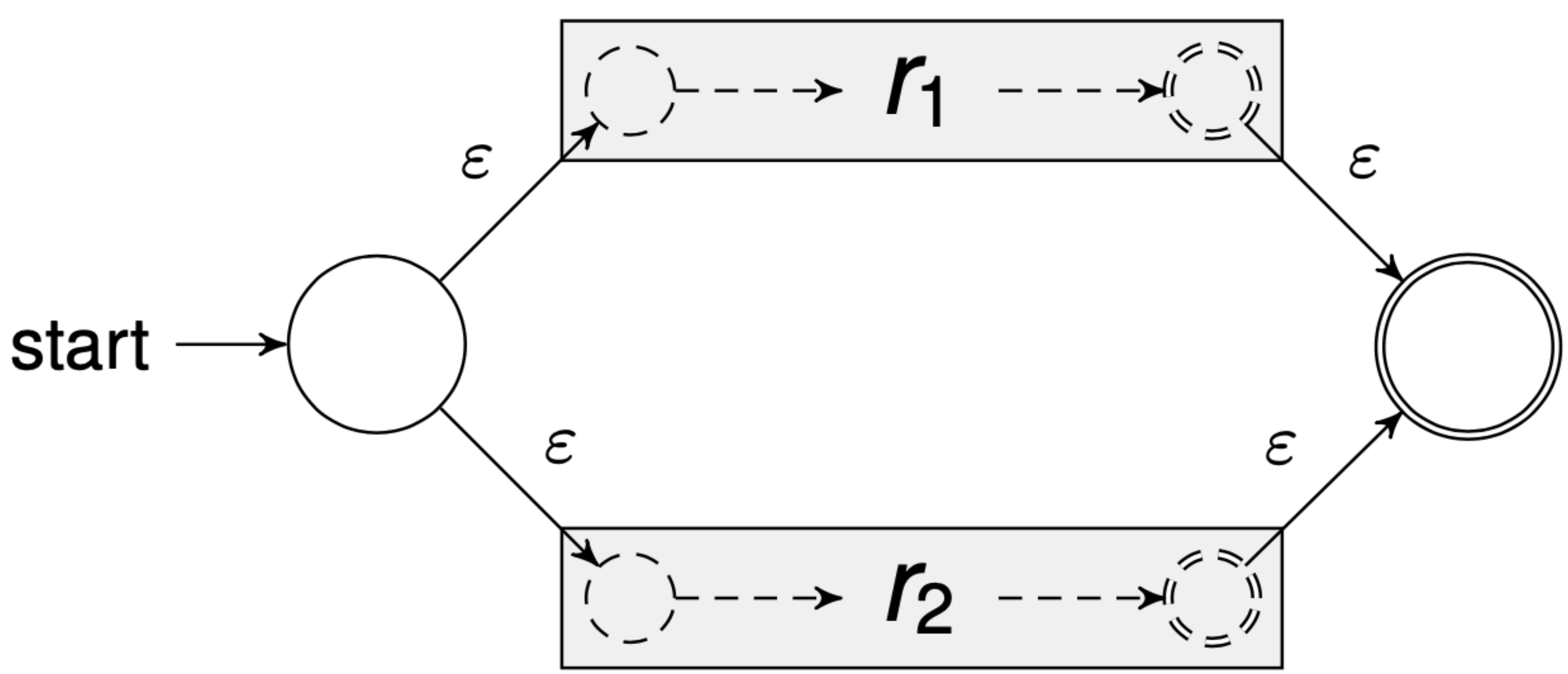
a



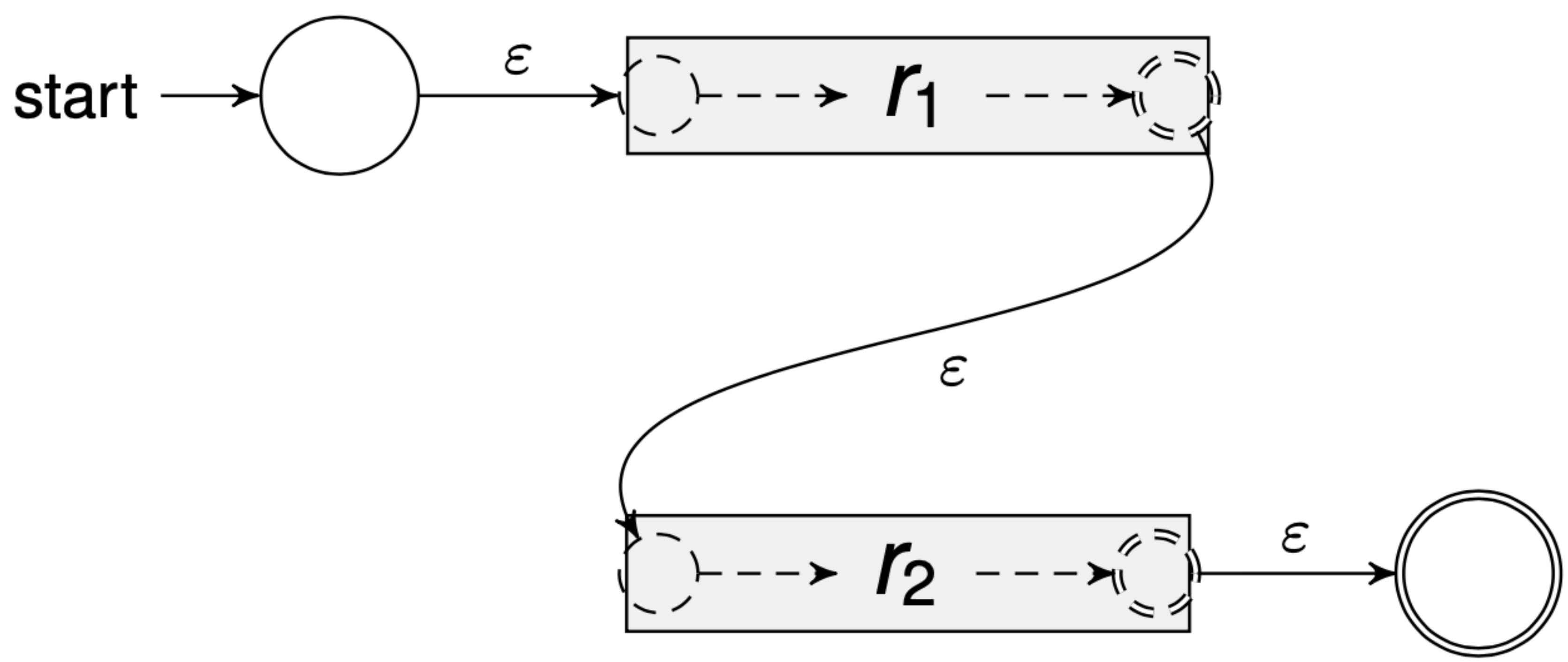
ε



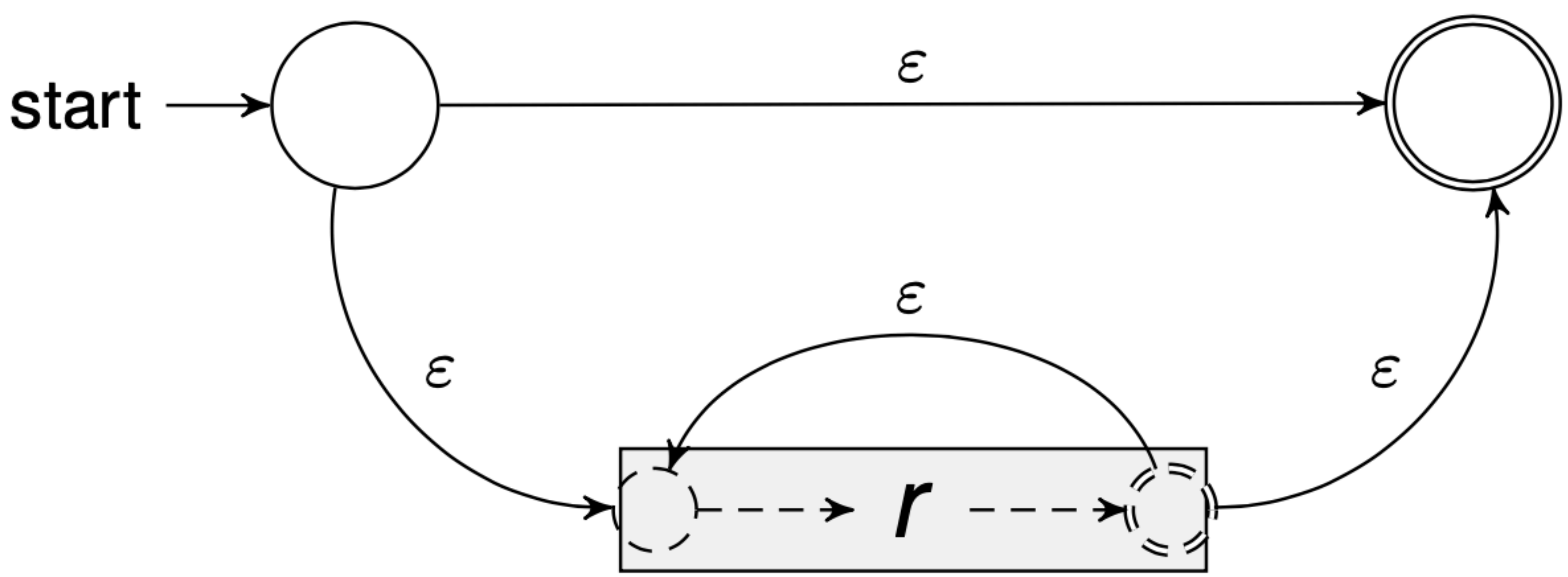
$$r_1 | r_2$$



$r_1 r_2$



r^*

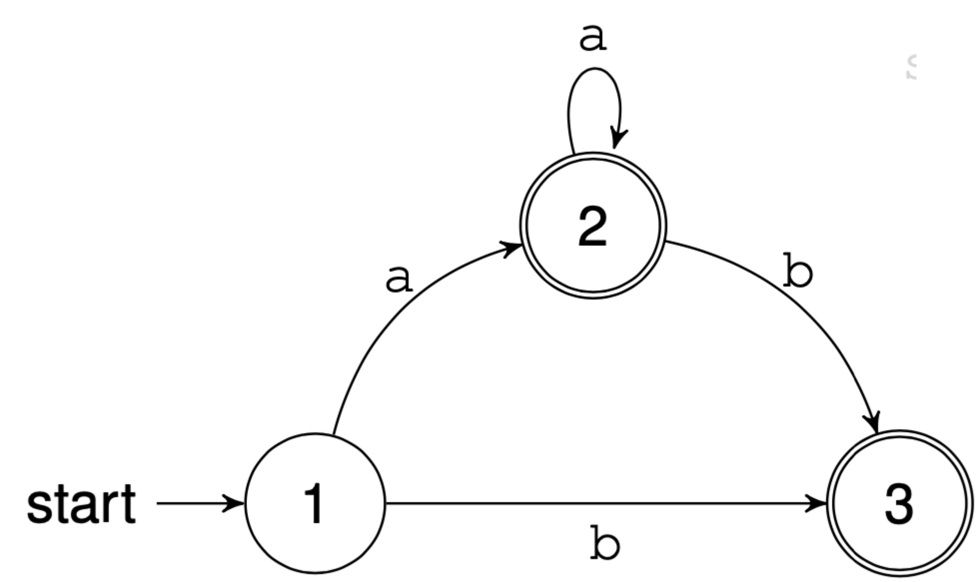


Exercise:

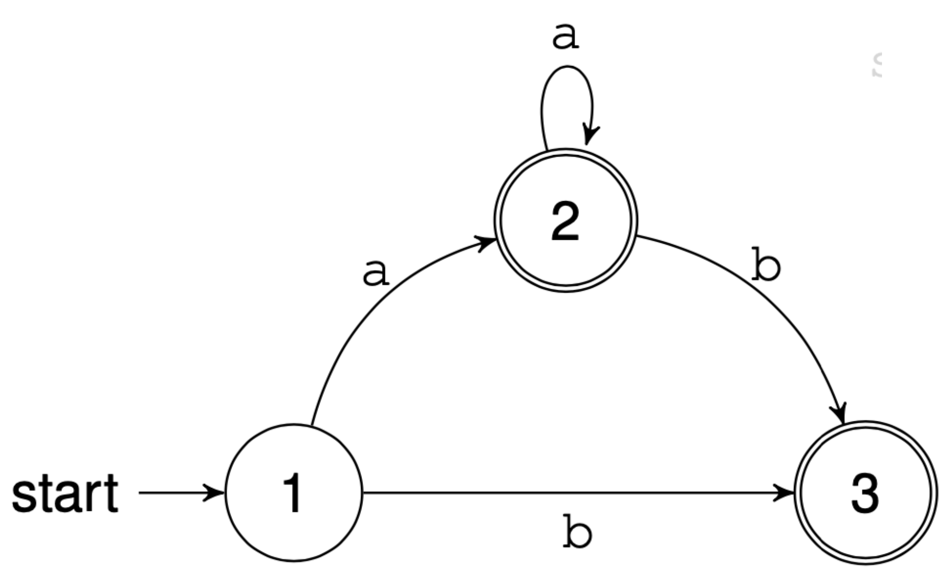
- *Make NFA for* `(ab)*`
- *Make NFA for* `(a|b)*`

Deterministic Finite Automata

- No ε-transitions
- Distinct transitions from each state
- Multiple accepting states OK



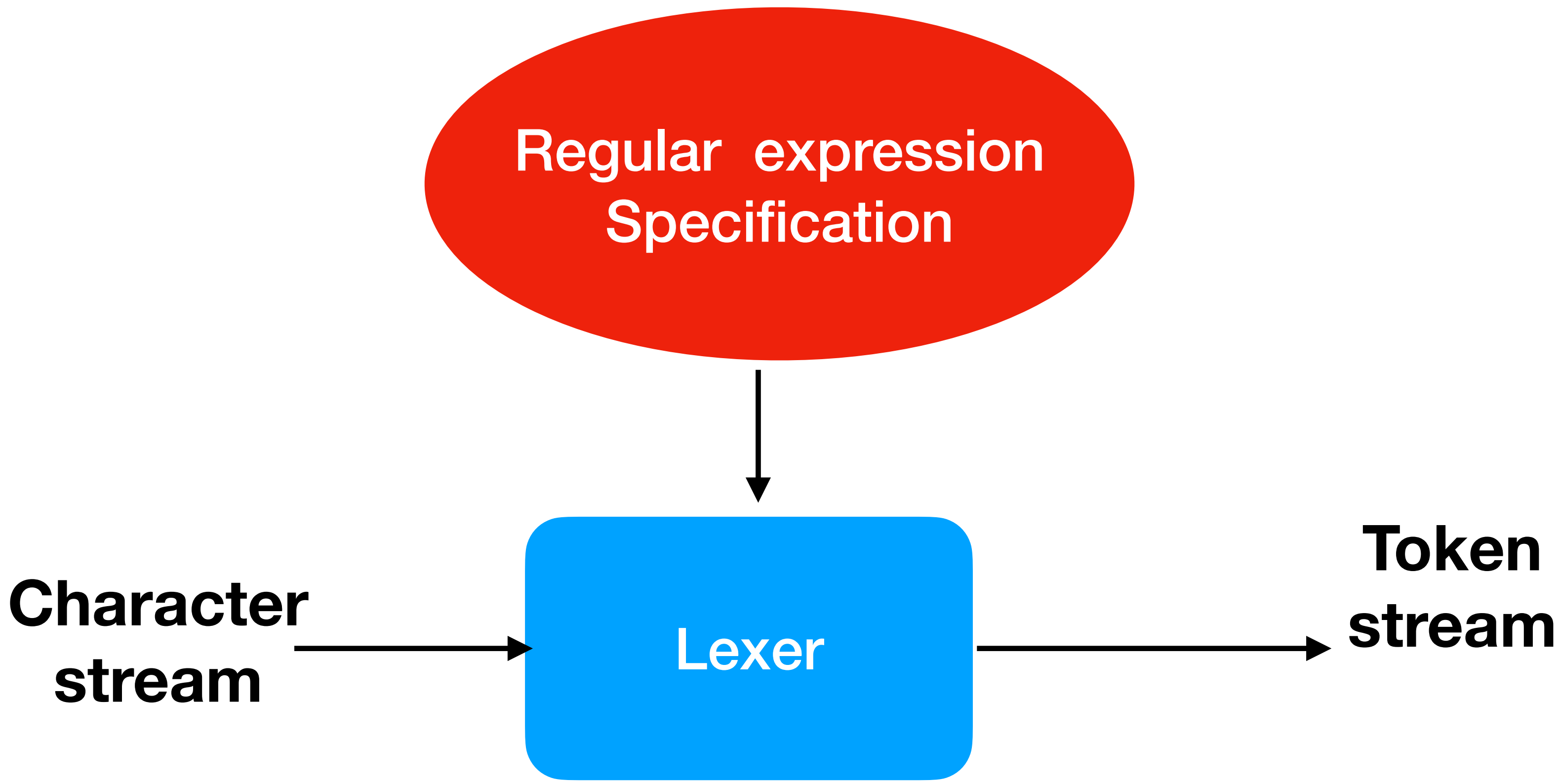
Why DFA



State	Input	Go to
1	a	2
1	b	3
2	a	2
2	b	3
3	a	fail
3	b	fail

- A DFA is easy to implement with table lookup:
    - next_state = table[current_state][next_symbol]
- Decides in linear time whether it accepts a string s
- For every NFA there is a corresponding DFA that accepts the same set of strings.

Summary


Regular expression
Specification
Character
stream
Lexer
Token
stream

- Basic Python:
  - https://colab.research.google.com/drive/15eiIquB2QVacfZWadm_jIw5Xv2ihl60J#scrollTo=UhcbBQUiStHG