Lecture B:

Languages, Automata, Regular Expressions & Scanners

CSE401/501m:

Introduction to Compiler Construction

Instructor: Gilbert Bernstein

Administrivia

- Read: textbook Ch 1, 2.1-2.4
- First homework out Thursday
  - ✦ Written problems on regexs/DFAs
  - ✦ We'll cover almost everything needed this week
  - ✦ Submit HW 1 on Gradescope
- Find a project partner if you haven't already
  - ✦ Be sure you agree on how you plan to share the work
  - ✦ We posted a form for **ONE** of you to send in partner info (Worth 1 point for both of you ***if*** done right) See calendar on webpage. Due by next Tuesday.
- Office hours have been posted on the calendar!

Administrivia (Friday)

- Read: textbook Ch 2.5
- First homework should be out
- Reminder: Project Partners are due next Tuesday

Outline

Review of Formal Languages, Grammars

Lexical Specification of Prog. Lang.

Regular Expressions

Finite Automata — Recognize Reg. Exp.

Scanners & Tokens

Outline

Review of Formal Languages, Grammars

Lexical Specification of Prog. Lang.

Regular Expressions

Finite Automata — Recognize Reg. Exp.

Scanners & Tokens

Programming Language Specifications

- Since the 1960s, the syntax of every significant programming language has been specified by a formal grammar
  - First done in 1959 using BNF (Backus-Naur Form); used to specify ALGOL 60 syntax
  - Borrowed from the Linguistics community (Chomsky)

*ALGOL 60 was adopted as the house style for pseudo-code algorithms published in CACM, the pre-eminent publication in Computer Science (Communications of the Association for Computing Machinery)

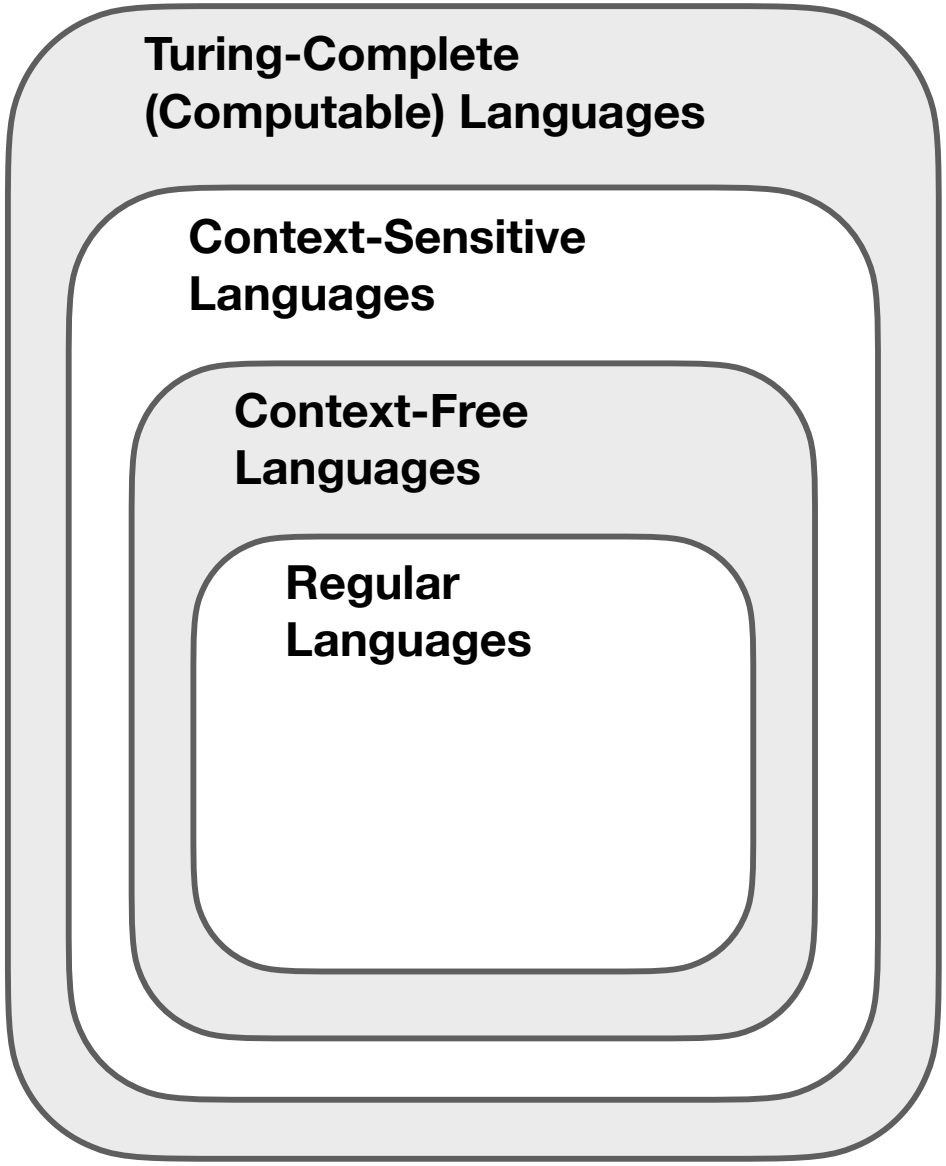
Formal Languages & Automata Theory (a review on one slide)

- **Alphabet** — a finite set of symbols and characters
- **String** — a finite, possibly empty sequence of symbols from an alphabet
- **Language** — a set of strings (possibly empty or infinite)
- An infinite language can be ***specified*** finitely
  - **Automaton** — a recognizer; a machine that accepts an input string if it is in the language (otherwise, rejects it)
  - **Grammar** — a generator; a system for producing all strings in the language (and no other strings)
- A particular language may be specified by many different grammars and automata
- A grammar or automaton specifies only one language

Language (Chomsky) Hierarchy

Quick Reminder

- **Regular languages** are specified by **regular expressions/grammars** and **finite automata (FSAs)**
  - ✦ Specs & implementations of scanners
- **Context-free languages** are specified by **context-free grammars** and **pushdown automata (PDAs)**
  - ✦ Specs & implementations of parsers
- Context-sensitive languages… aren't too important (for us)
- **Recursively-enumerable languages** are specified by **general grammars** and **Turing machines**



Example

Grammar for a Tiny Language

$$
\begin{aligned}
program &::= statement \mid program\ statement \\
statement &::= assignStmt \mid ifStmt \\
assignStmt &::= id\ \underline{\texttt{=}}\ expr\ \underline{\texttt{;}} \\
ifStmt &::= \underline{\texttt{if}}\ \underline{\texttt{(}}\ expr\ \underline{\texttt{)}}\ statement \\
expr &::= id \mid int \mid expr\ \underline{\texttt{+}}\ expr \\
id &::= \underline{\texttt{a}} \mid \underline{\texttt{b}} \mid \underline{\texttt{c}} \mid \underline{\texttt{i}} \mid \underline{\texttt{j}} \mid \underline{\texttt{k}} \mid \underline{\texttt{n}} \mid \underline{\texttt{x}} \mid \underline{\texttt{y}} \mid \underline{\texttt{z}} \\
int &::= \underline{\texttt{0}} \mid \underline{\texttt{1}} \mid \underline{\texttt{2}} \mid \underline{\texttt{3}} \mid \underline{\texttt{4}} \mid \underline{\texttt{5}} \mid \underline{\texttt{6}} \mid \underline{\texttt{7}} \mid \underline{\texttt{8}} \mid \underline{\texttt{9}}
\end{aligned}
$$

Exercise

Derive a simple program

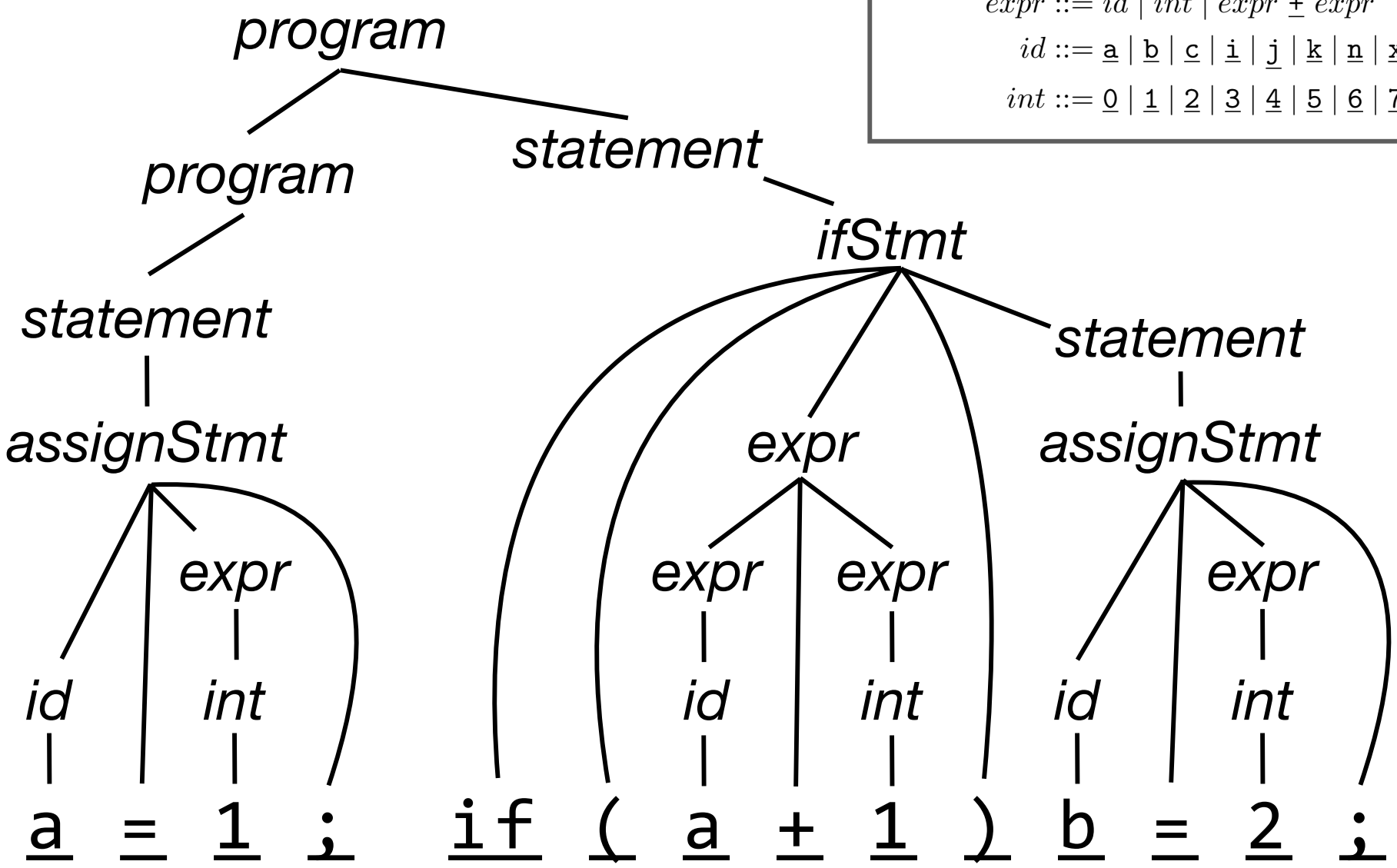
$$
\begin{aligned}
program &::= statement \mid program\ statement \\
statement &::= assignStmt \mid ifStmt \\
assignStmt &::= id\ \texttt{=}\ expr\ \texttt{;} \\
ifStmt &::= \texttt{if}\ \texttt{(}\ expr\ \texttt{)}\ statement \\
expr &::= id \mid int \mid expr\ \texttt{+}\ expr \\
id &::= \texttt{a} \mid \texttt{b} \mid \texttt{c} \mid \texttt{i} \mid \texttt{j} \mid \texttt{k} \mid \texttt{n} \mid \texttt{x} \mid \texttt{y} \mid \texttt{z} \\
int &::= \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}
\end{aligned}
$$

```
a = 1 ;  if ( a + 1 ) b = 2 ;
```

Exercise

Derive a simple program

$$
\begin{aligned}
program &::= statement \mid program\ statement \\
statement &::= assignStmt \mid ifStmt \\
assignStmt &::= id\ \texttt{=}\ expr\ \texttt{;} \\
ifStmt &::= \texttt{if}\ \texttt{(}\ expr\ \texttt{)}\ statement \\
expr &::= id \mid int \mid expr\ \texttt{+}\ expr \\
id &::= \texttt{a} \mid \texttt{b} \mid \texttt{c} \mid \texttt{i} \mid \texttt{j} \mid \texttt{k} \mid \texttt{n} \mid \texttt{x} \mid \texttt{y} \mid \texttt{z} \\
int &::= \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}
\end{aligned}
$$



Productions

$$
\begin{aligned}
program &::= statement \mid program\ statement \\
statement &::= assignStmt \mid ifStmt \\
assignStmt &::= id\ \texttt{=}\ expr\ \texttt{;} \\
ifStmt &::= \texttt{if}\ \texttt{(}\ expr\ \texttt{)}\ statement \\
expr &::= id \mid int \mid expr\ \texttt{+}\ expr \\
id &::= \texttt{a} \mid \texttt{b} \mid \texttt{c} \mid \texttt{i} \mid \texttt{j} \mid \texttt{k} \mid \texttt{n} \mid \texttt{x} \mid \texttt{y} \mid \texttt{z} \\
int &::= \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}
\end{aligned}
$$

- The rules of a grammar are called **productions**
- Rules contain
  - **non-terminal symbols** — the variables of the grammar (e.g. *program*, *statement*, *id*, etc.)
  - **terminal symbols** — concrete syntax that appears in programs (e.g. `a`, `b`, `c`, `0`, `1`, `if`, `+`, `=`, etc.)
- The meaning of a production is that *in a derivation* a non-terminal (occurring on the left-hand-side of the production) may be replaced by the sequence of terminals and non-terminals occurring to the right. (we just saw this)
- There is often a choice (e.g. *assignStmt* | *ifStmt*) of which rule to expand with. Thus, grammar derivations are non-deterministic in general.

Alternative Notations

for productions

- There are several notations for productions in common use; all mean the same thing

$$\textit{ifStmt} ::= \underline{\texttt{if}}\ \underline{(}\ \textit{expr}\ \underline{)}\ \textit{statement}$$

$$\textit{ifStmt} ::= \text{if}\ (\ \textit{expr}\ )\ \textit{statement}$$

$$\textit{ifStmt} \rightarrow \text{if}\ (\ \textit{expr}\ )\ \textit{statement}$$

$$\langle \text{ifStmt} \rangle ::= \text{if}\ (\ \langle \text{expr} \rangle\ )\ \langle \text{statement} \rangle$$

- Note: ***concrete syntax*** (keywords/tokens like `if`) as opposed to ***meta syntax*** (variables like *expr*)
- AND there is meta-meta-syntax

Recognizing Levels of Notation

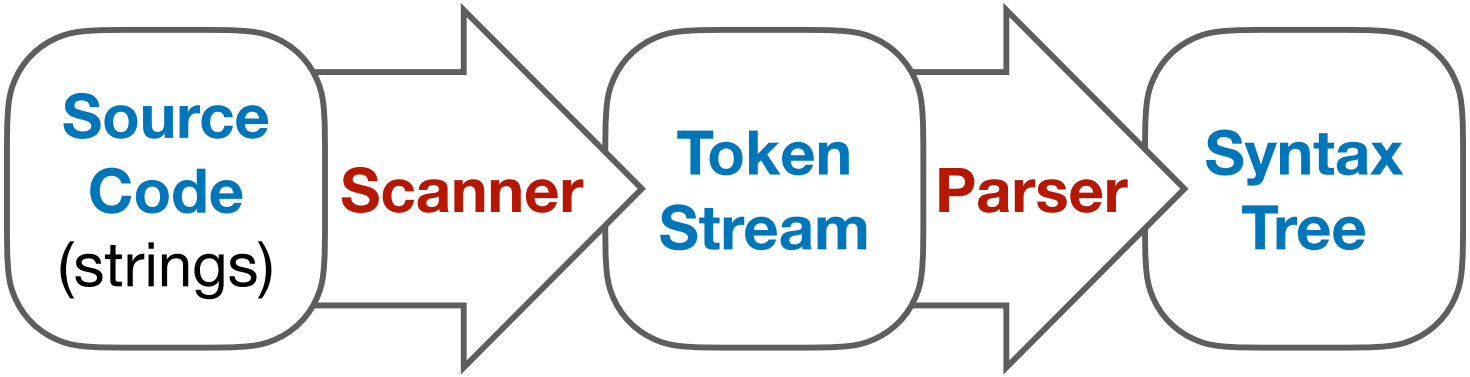
- Note the difference between ***concrete syntax*** (keywords/tokens like `if`) as opposed to ***meta syntax*** (variables like *expr*)
  - e.g. ***ifExpr ::= `if` `(` expr `)` statement***
- Like with learning algebra, you learn to read structure
  - e.g. 3 + 4 • x - y • 28
- One can even specify the general syntax of productions!
  - i.e. α ::= $\beta_0\beta_1\ldots\beta_n$

Parsing

- **Parsing** is the process of reconstructing the derivation (syntactic structure) of some source program (i.e. string)
- In principle a single recognizer could work directly from a concrete, character-by-character grammar
- In practice, this is (almost) never done

Parsing & Scanning

- In real compilers, the recognizer is split in two phases
  - **Scanner** — translate input characters to tokens
    - *also report lexical errors like illegal characters and illegal symbols; skip past things with no semantic meaning in the language, like comments and whitespace (in most languages)*
  - **Parser** — read token stream and reconstruct the derivation



Why Separate the Scanner & Parser?

- Simplicity & Separation of Concerns
  - ✦ Scanner hides details from parser (comments, whitespace, input files, etc.)
  - ✦ Parser becomes easier to build; has simpler input stream (tokens) and simpler interface for input
- Efficiency
  - ✦ Scanner recognizes regular expressions — proper subset of context free grammars
    - but still consumes a surprising amount of the total execution time — e.g. suppose input file has 10,000 characters, but only 1,000 post-scan tokens

But…

- Not always possible to separate cleanly
- e.g. C/C++/Java type vs. identifier
  - ✦ Parser would like to know which names are types vs. identifiers, but…
  - ✦ Scanner does not know how things are declared
- So we hack around it somehow
  - ✦ Either use simpler grammar and disambiguate later, or communicate between scanner & parser
  - ✦ Engineering issue — try to keep interfaces as simple & clean as possible

Outline

Review of Formal Languages, Grammars

Lexical Specification of Prog. Lang.

Regular Expressions

Finite Automata — Recognize Reg. Exp.

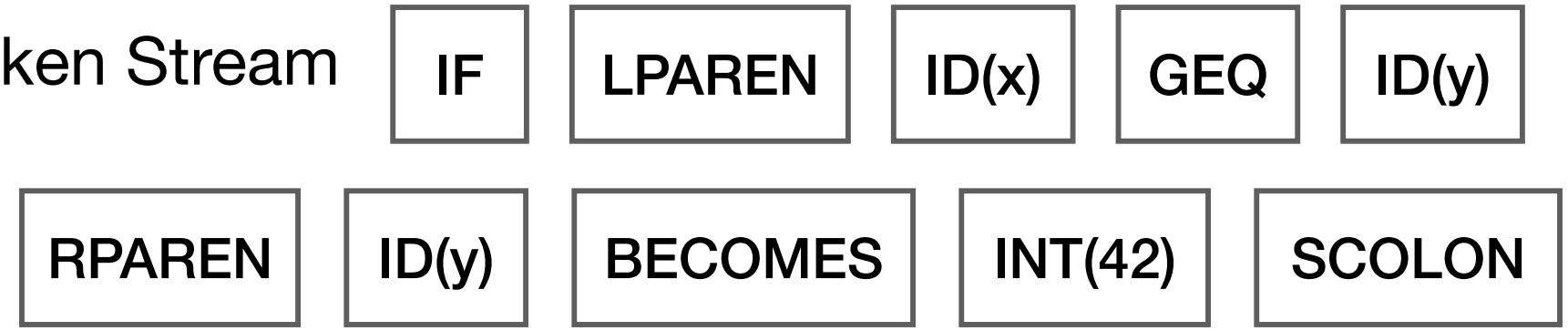
Scanners & Tokens

Scanner Example

- Input text

```
// this statement does very little
if (x >= y) y = 42;
```

- Token Stream



  - ✦ Note: tokens are atomic items, ***not*** character strings; comments & whitespace are not tokens (in most languages — counterexamples include Python indenting, Ruby and JavaScript newlines)
    - Token objects sometimes carry associated data (e.g. numeric value or variable name)

Typical Tokens in Prog. Lang.

- Operators & Punctuation
  - e.g. `+ - * / ( ) { } [ ] ; : :: < <= == = != ! ...`
  - each is a distinct lexical class
- Keywords
  - e.g. `if while for goto return switch void ...`
  - each is a distinct lexical class (*not* a string)
- Identifiers
  - A single ID lexical class, but parameterized by actual id string
- Integer constants
  - A single INT lexical class, but parameterized by int value
- Other constants, etc.

Principle of Longest Match

- In most languages, the scanner should pick the longest possible string as the next token if there is a choice
- Example

```
return maybe != iffy;
```

should be recognized as 5 tokens

RETURN	ID(maybe)	NEQ	ID(iffy)	SCOLON

i.e. `!=` is one token, not two; `iffy` is an ID, not IF followed by ID(fy)

Lexical Complications

- Most modern languages are free-form
  - ✦ layout doesn't matter
  - ✦ whitespace separates tokens
- Alternatives / Variations
  - ✦ Fortran — line oriented
  - ✦ Haskell, Python — indentation and layout implies grouping
  - ✦ Ruby, Javascript — newlines can end statements
- And other confusions
  - ✦ C++, Java — is >> a shift operator or the closing of two nested templates/generics?

Outline

Review of Formal Languages, Grammars

Lexical Specification of Prog. Lang.

Regular Expressions

Finite Automata — Recognize Reg. Exp.

Scanners & Tokens

Regular Expressions and FAs

- The lexical grammar (structure) of most programming languages can be specified with regular expressions

  (ok, maybe a *little* cheating is needed)

- Tokens can be recognized by a deterministic finite automaton
  - ✦ The automaton can either be table-driven (generated from a specification) or hand-written, based on a lexical grammar (i.e. a regular expression)

Regular Expressions

- Defined over some **alphabet** Σ
  - ✦ For programming languages, this alphabet is usually ASCII or Unicode
- Aside — recall that Σ^* is the set of all strings (potentially empty) with characters from the alphabet Σ
- If α is a regular expression, then $L(\alpha)$ is the language (set of strings; i.e. subset of Σ^*) ***generated*** by α

Primitive REs

α	$L(\alpha)$	Notes
c	$\{c\}$	Singleton set, for each c in Σ
ϵ	$\{\epsilon\}$	Empty string
$\varnothing$	$\{\}$	Empty language

Operations on REs

α	$L(\alpha)$	Notes
$\alpha\beta$	$\{s_1 s_2 \mid s_1 \in L(\alpha),\ s_2 \in L(\beta)\}$	*Concatenation*
$\alpha \mid \beta$	$L(\alpha) \cup L(\beta)$	*Combination (union)*
α^*	$\{s_1 \cdots s_n \mid s_i \in L(\alpha),\ n \geq 0\}$	*0 or more occurrences (Kleene closure)*

- precedence: * (highest), concatenation, | (lowest)
- parentheses can be used to group REs as needed
- On computer need a way to escape * and | (don't worry on paper)

Examples

Reg. Exp.	Meaning
`+`	single + character
`!`	single ! character
`=`	single = character
`!=`	2 character sequence `!=`
`xyzzy`	5 character sequence `xyzzy`
`(1|0)*`	0 or more binary digits (i.e. seq of `0`s, `1`s)
`(1|0)(1|0)*`	1 or more binary digits (i.e. seq of `0`s, `1`s)
`0|1(1|0)*`	sequence of binary digits with no leading `0`s, except for `0` by itself

Abbreviations

- The basic operations generate all possible regular expression, but there are common abbreviations used for convenience. Some examples:

Abbr.	Meaning	Notes
$\alpha+$	$\alpha\alpha^*$	1 or more occurrences
$\alpha?$	$(\alpha \mid \epsilon)$	0 or 1 occurrences
[a-z]	(a|b|…|z)	1 character in given range
[abxyz]	(a|b|x|y|z)	1 of the given characters

More Examples

Reg. Exp.	Meaning
`[abc]+`	
`[abc]*`	
`[0-9]+`	
`[1-9][0-9]*`	
`[a-zA-Z][a-zA-Z0-9_]*`	

More Examples

Reg. Exp.	Meaning
`[abc]+`	sequence of 1 or more as, bs, cs
`[abc]*`	sequence of 0 or more as, bs, cs
`[0-9]+`	sequence of 1 or more decimal digits
`[1-9][0-9]*`	sequence of 1 or more decimal digits (without a leading 0)
`[a-zA-Z][a-zA-Z0-9_]*`	Identifiers in your *Favorite Programming Language*™

Abbreviations

- Many systems allow abbreviations to make writing and reading definitions or specifications easier

$$name ::= \alpha$$

  - ✦ Key restriction! Definitions must not be circular (recursive) directly or indirectly (otherwise the resulting language might not be regular)

Example

- Possible syntax for numeric constants

$$digit ::= [\underline{0} - \underline{9}]$$
$$digits ::= digit+$$
$$number ::= digits\ (\underline{.}\ digits)?([\underline{\texttt{eE}}](\underline{+}|\underline{-})?\ digits)?$$

- How would you describe this set in English?
- What are some examples of legal constants (strings) generated by *number* ?
  - ✦ What are the differences between these and numeric constants in YFPL?
    (Your Favorite Programming Language)

Outline

Review of Formal Languages, Grammars

Lexical Specification of Prog. Lang.

Regular Expressions

Finite Automata — Recognize Reg. Exp.

Scanners & Tokens

Recognizing regular languages

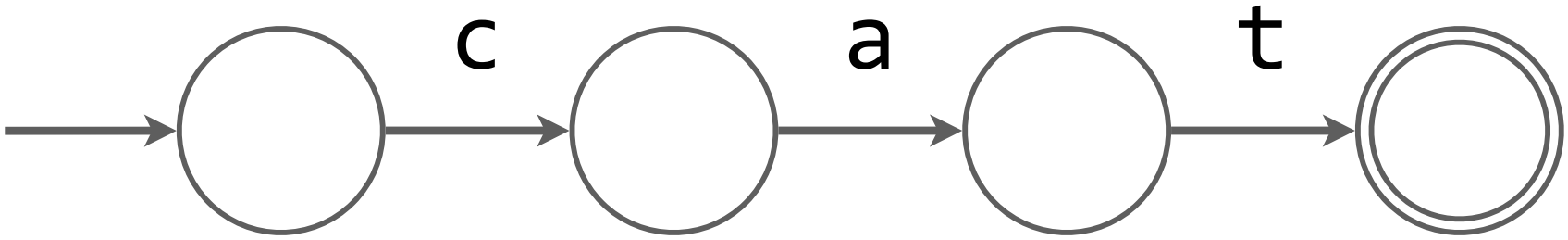
- **Finite automata** can be used to recognize strings generated by regular expressions
- Can write by hand or generate automatically
  - Reasonably straightforward, and can be done systematically
  - Tools like Lex, Flex, JFlex (etc.) do this automatically, given a set of regular expressions
  - Same technique used in grep, sed, text editors, other regular expression packages/tools

Finite State Automaton (FSA)

A review on one slide

- A finite set of **states** (S)
  - One marked as **initial state** (s_0)
  - One or more marked as **final states** ($F \subseteq S$)
- A set of **transitions** from state to state
  - equivalently, a function that outputs the *set of possible next states* starting from a *current state* and *input character* (formally $\delta : (\Sigma \times S) \rightarrow \mathscr{P}(S)$)
  - often depicted as a set of Σ-labeled graph edges
- Operate by reading input symbols/characters and transitioning to some valid state
  - When drawing graphs, can include ϵ-labeled transition edges, that can be taken without consuming an input character
- **Accept** if (for some execution) when there is no more input, the state is final
  - More involved in a scanner because (1) there are multiple kinds of final state (i.e. tokens) and (2) we accept the longest prefix of the input that is accepted
- **Reject** if (1) no transition possible, or (2) no more input and not in a final state (DFA)
  - Some versions (including textbook) have an explicit "error" state; transition to it when no other transition possible; better to omit/special-case this for CSE 401

Example: FSA for “cat”

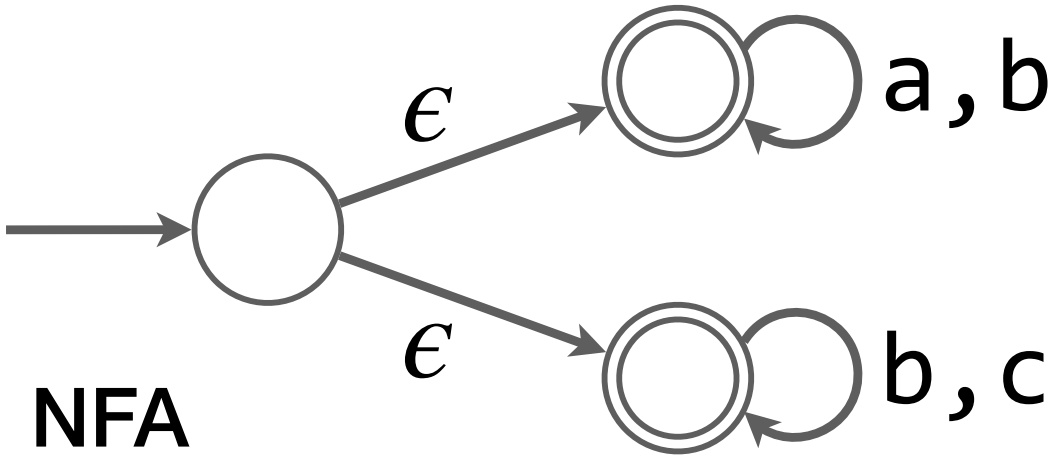


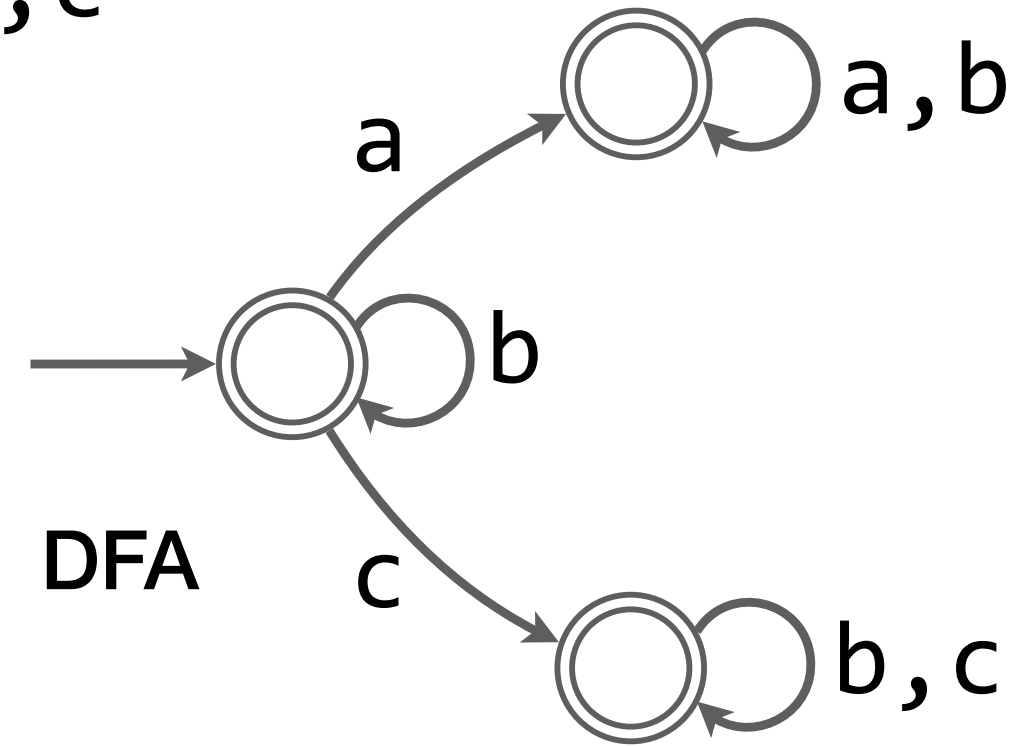
DFA vs. NFA (determinism, or not?)

- **Deterministic** Finite Automata (**DFA**)
  - At most one state ($\delta(a, s)$) the FA can transition to on a given input from a given state (zero states if "error")
  - No ϵ-labeled edges allowed in graph representation
- **Non-deterministic** Finite Automata (**NFA**)
  - There is some input and state on which the FA can transition to more than one state; i.e. there are non-deterministic choices
  - **Accept** if there is some seq. of choices reaching a final state
  - **Reject** if all possible choices fail to reach a final state
  - When simulating, this requires guessing and backtracking

DFA vs. NFA example

(a|b)*|(b|c)*





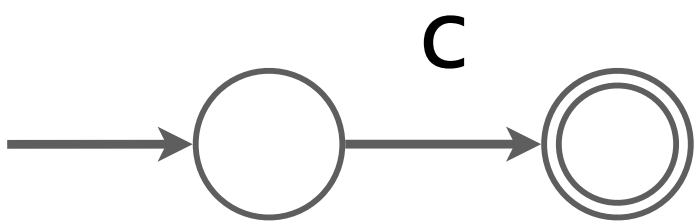
Building DFAs from REs

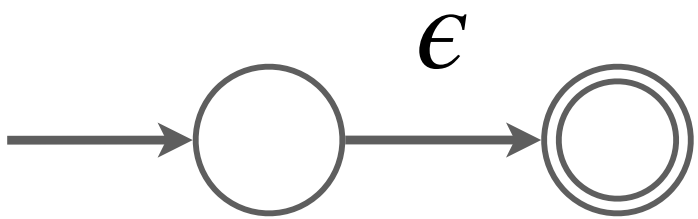
- We want DFAs for speed (no backtracking or guessing)
- But conversion from REs to NFAs is much simpler (e.g. the example on the last slide)
- Our approach will be RE → NFA → DFA
  - The second step of NFA → DFA will be done by something called the "subset construction"

RE → NFA (Recursion)

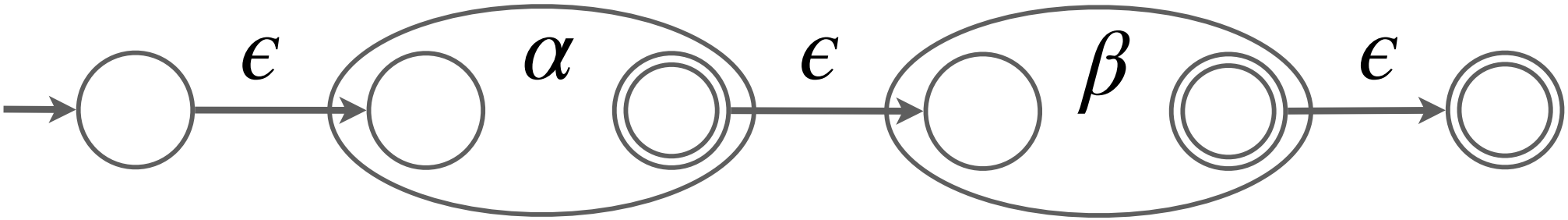
- Recall that a regular expression is either
  - c, ϵ (base cases)
  - $\alpha\beta$, $\alpha \mid \beta$, $\alpha*$ (recursive/inductive cases)
- Structural Induction
  - Most code/algorithms in this class will be structurally inductive!
  - Specify how to construct an NFA for each base case
  - Specify how to construct an NFA for each inductive case, given an NFA for each sub-expression

RE → NFA: base cases

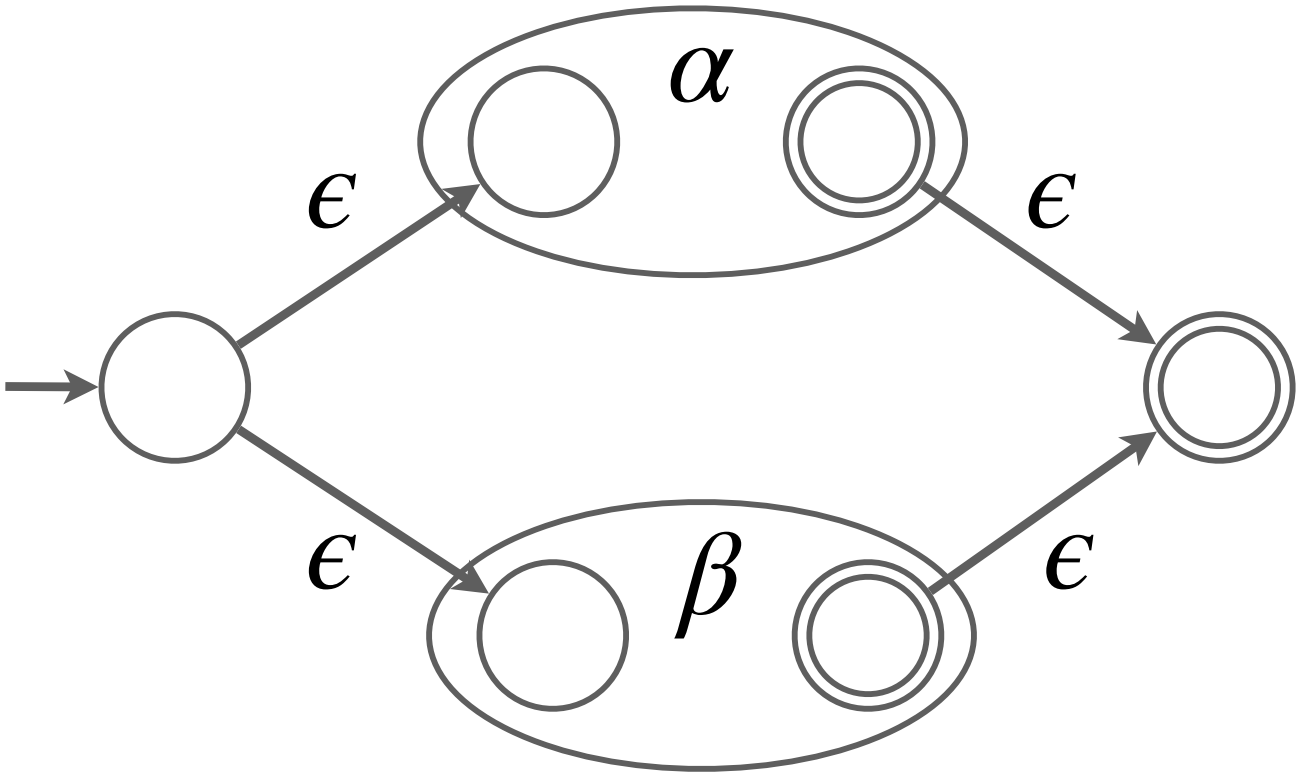




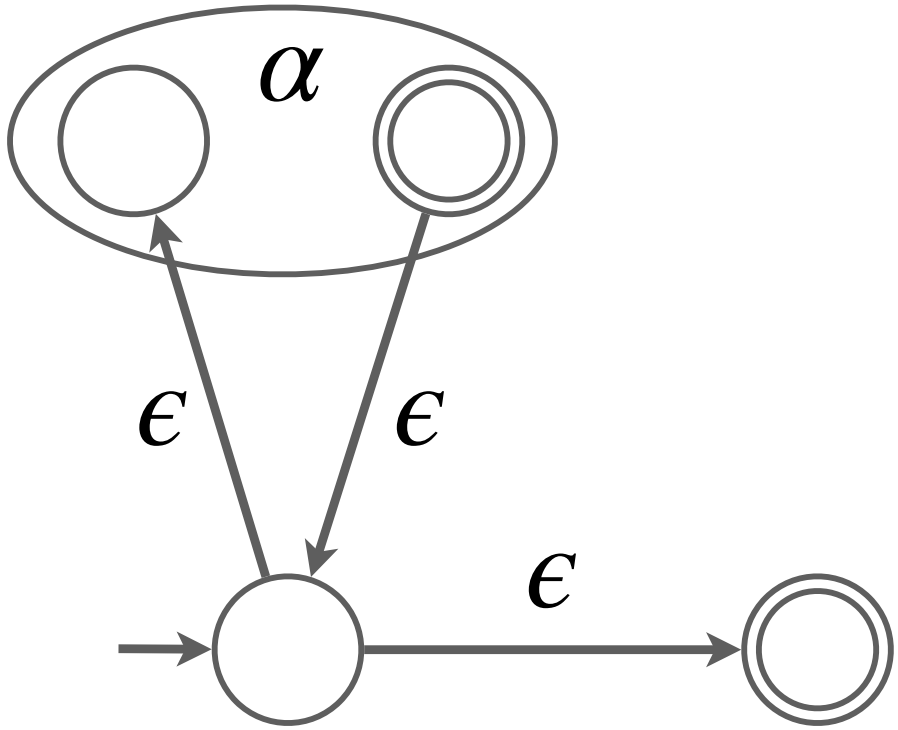
RE → NFA: concatenation



RE → NFA: union



RE → NFA: Kleene closure

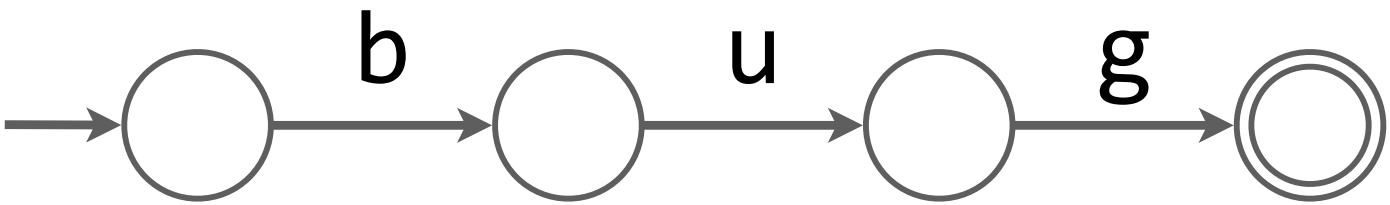


Exercise: Draw the NFA

b(at|ag)|bug

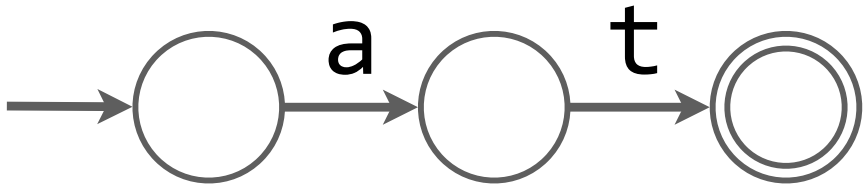
Exercise: Draw the NFA

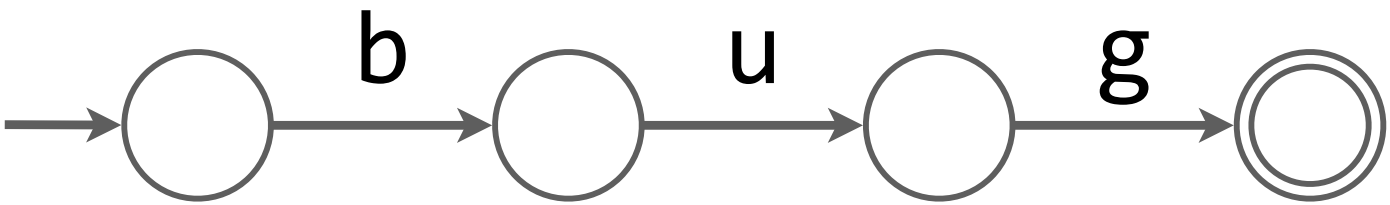
b(at|ag)|bug



Exercise: Draw the NFA

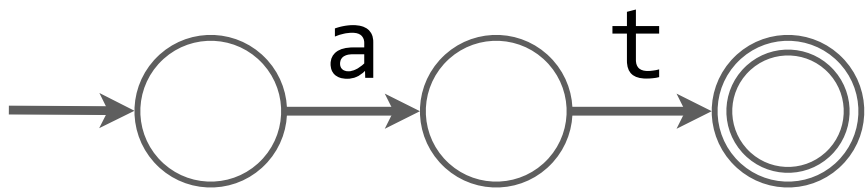
b(at|ag)|bug

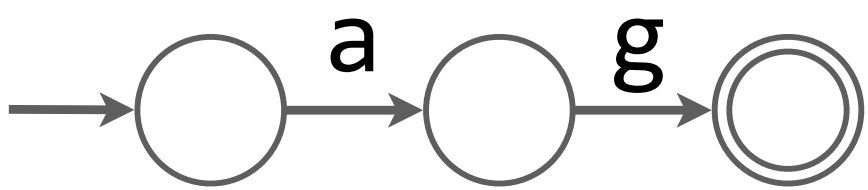


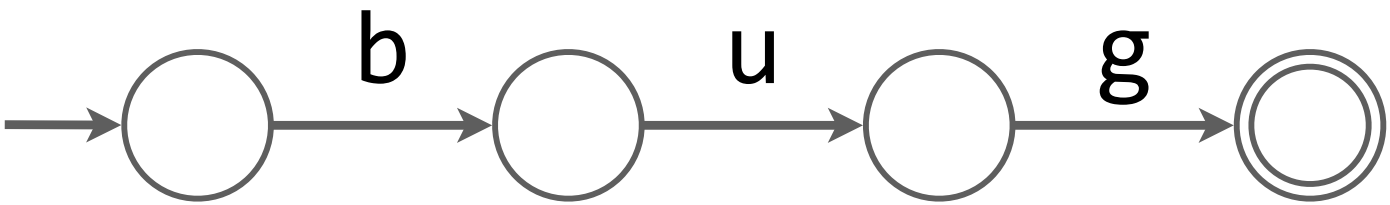


Exercise: Draw the NFA

b(at|ag)|bug

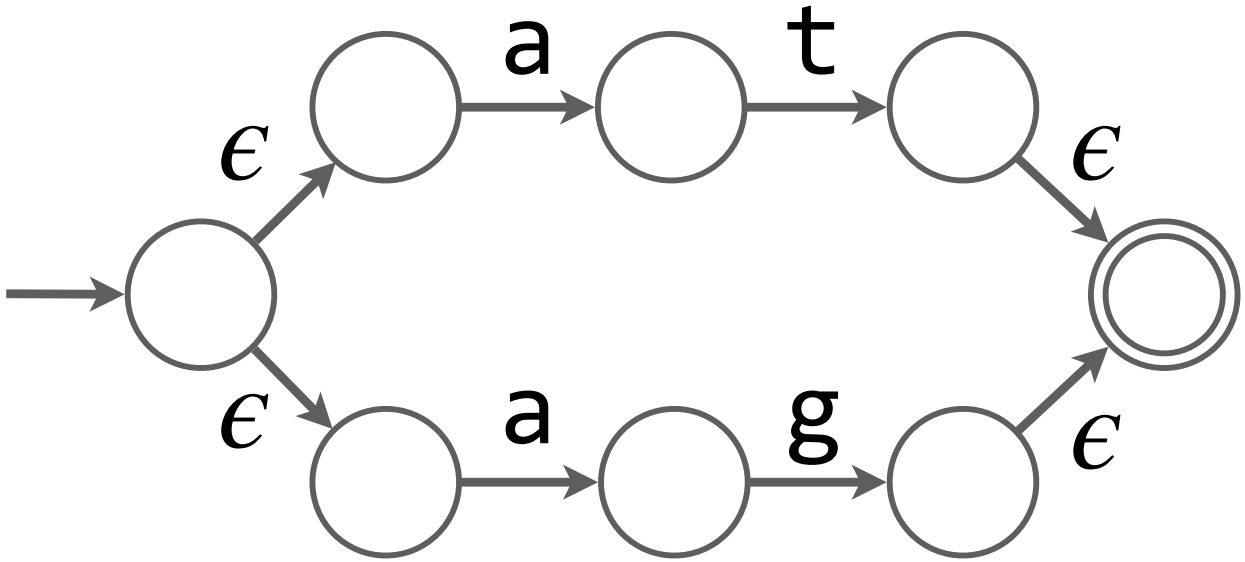


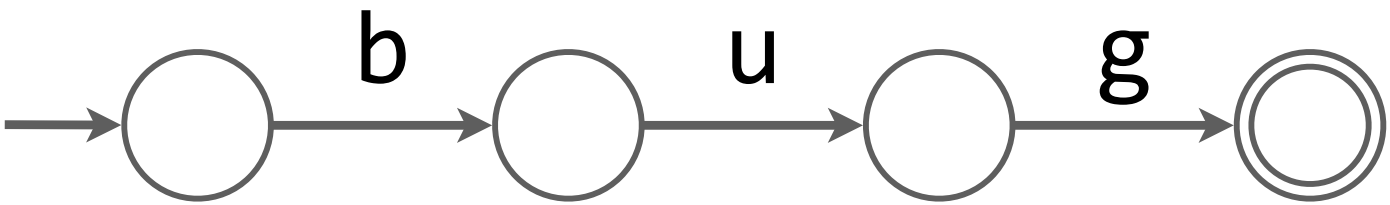




Exercise: Draw the NFA

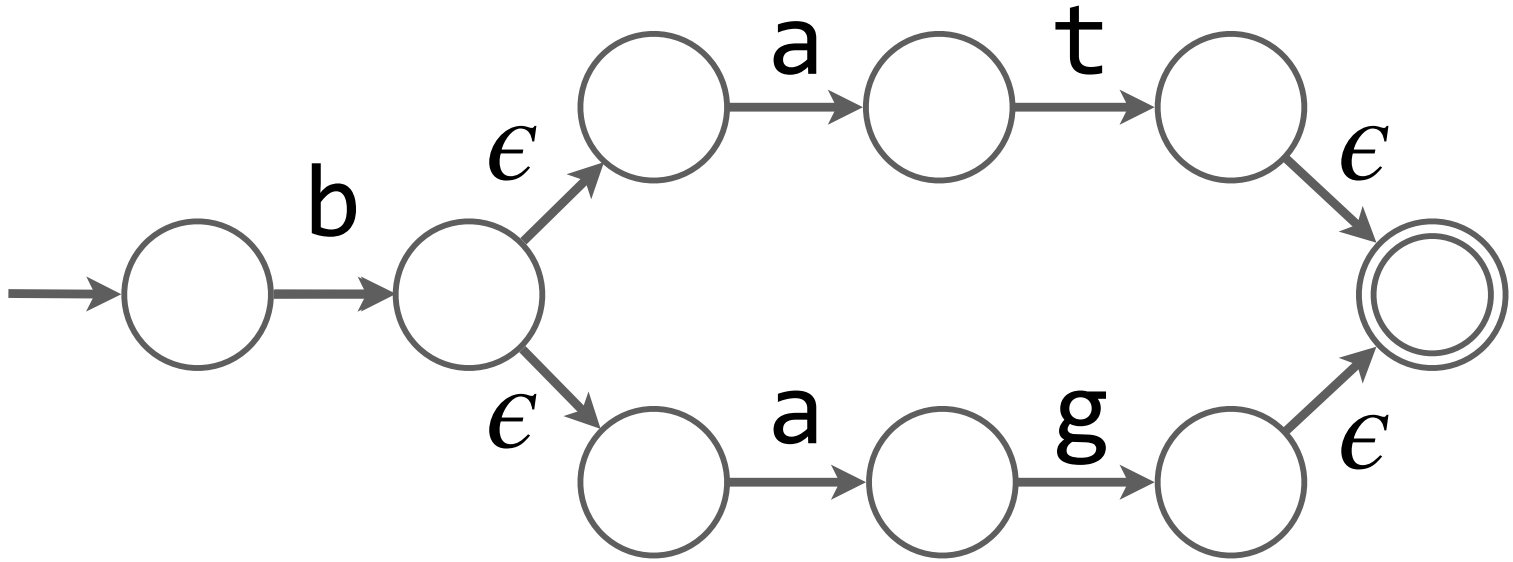
b(at|ag)|bug

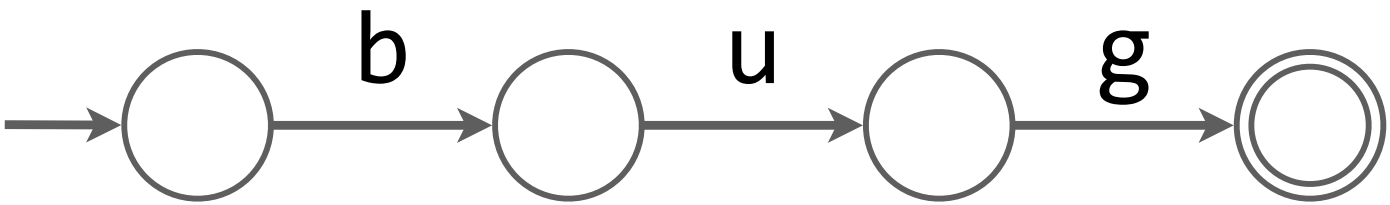




Exercise: Draw the NFA

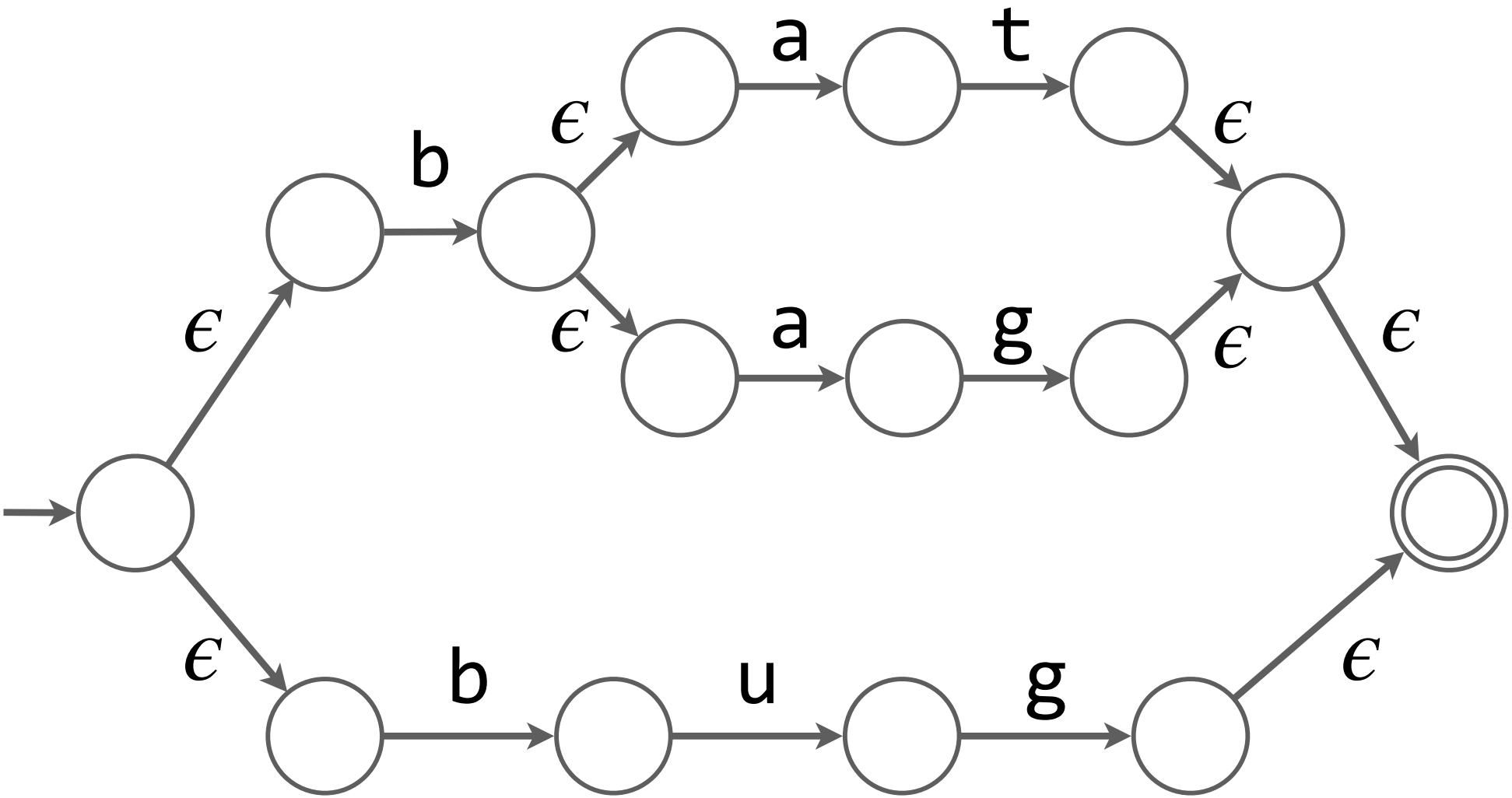
b(at|ag)|bug





Exercise: Draw the NFA

b(at|ag)|bug



NFA → DFA

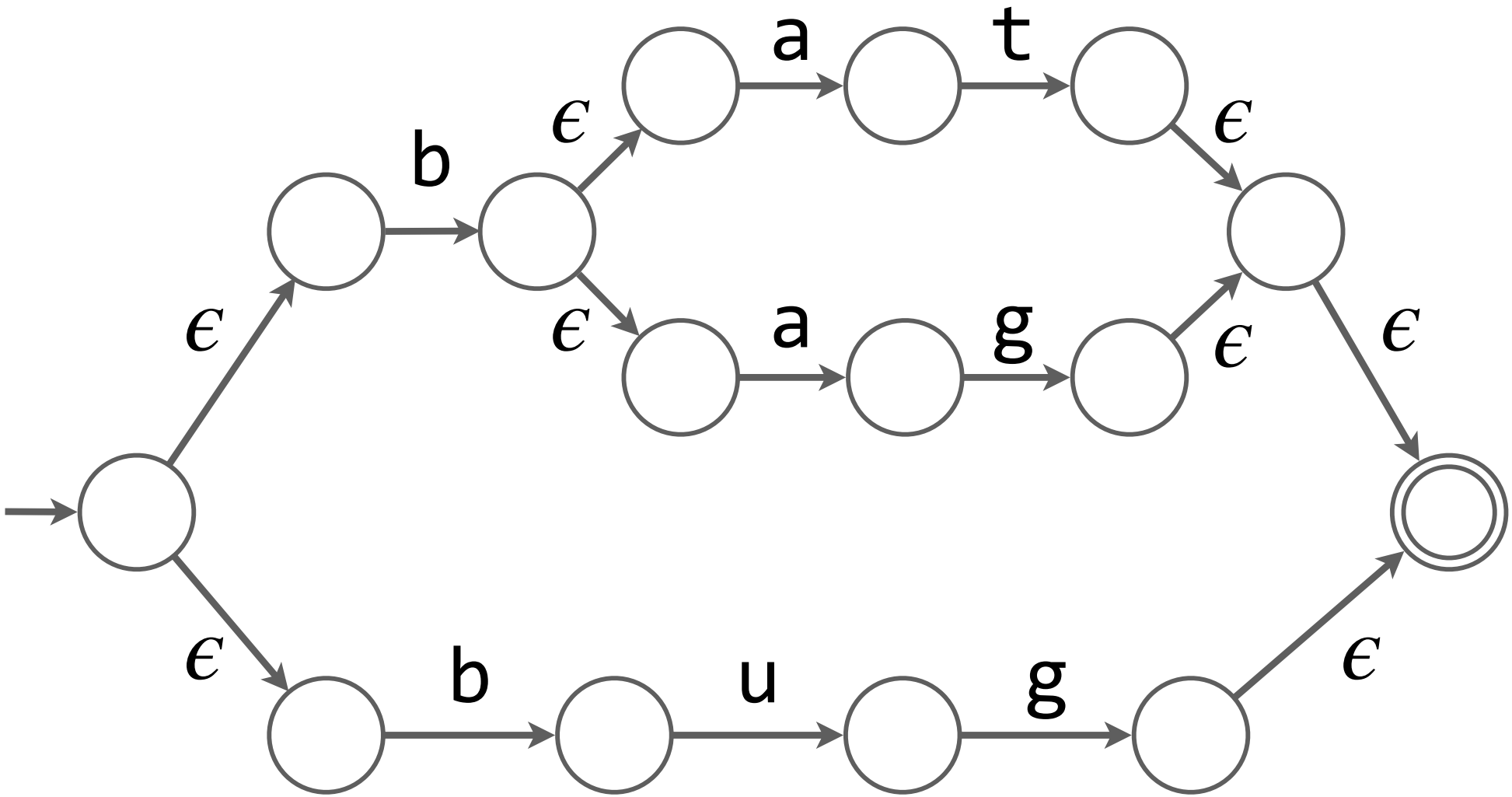
***Formally**
note: not required for HW or Exams

$$\delta_{\mathscr{D}}(c, X) = \bigcup_{x \in X} \delta_{\mathscr{N}}(c, x)$$

- Subset Construction
  - Take an input NFA $\mathscr{N}$ and return a DFA $\mathscr{D}$
  - DFA $\mathscr{D}$ **states** correspond to subsets of the states of NFA $\mathscr{N}$ $S_{\mathscr{D}} = \mathscr{P}(S_{\mathscr{N}})$ (so naively, 2^n DFA states if NFA has n states)
  - (Informal*) DFA $\mathscr{D}$ **transitions** are constructed by collecting the set of NFA states that can be reached after reading the same input
- Algorithm that usually avoids enumerating all 2^n possible states
  - example of a **fixed-point computation**; only add DFA states as we discover they are reachable
  - Resulting DFA may still have more states than needed
    - see textbooks for construction and minimization details

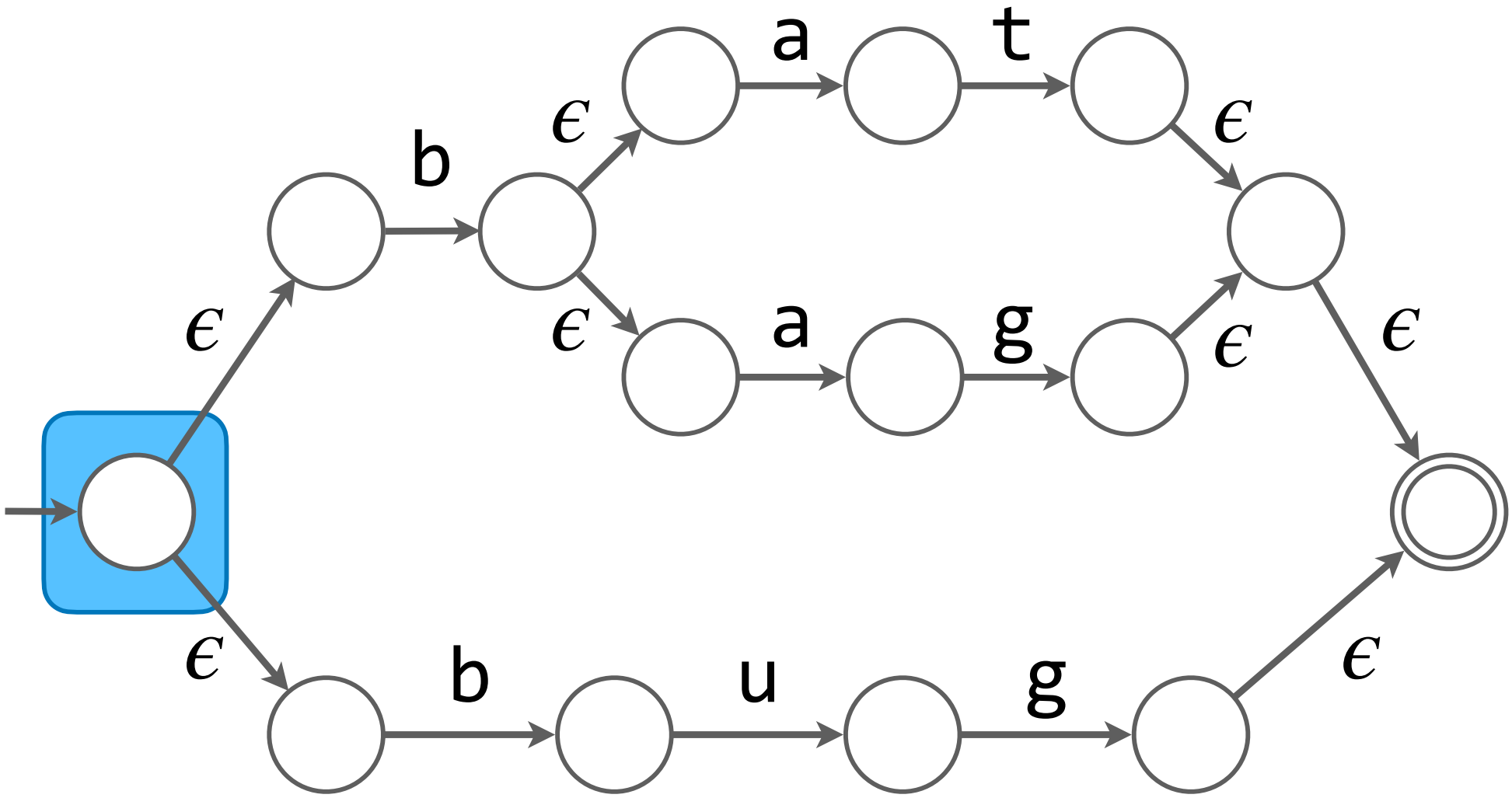
Exercise: (NFA → DFA)

build DFA for b(at|ag)|bug given the NFA



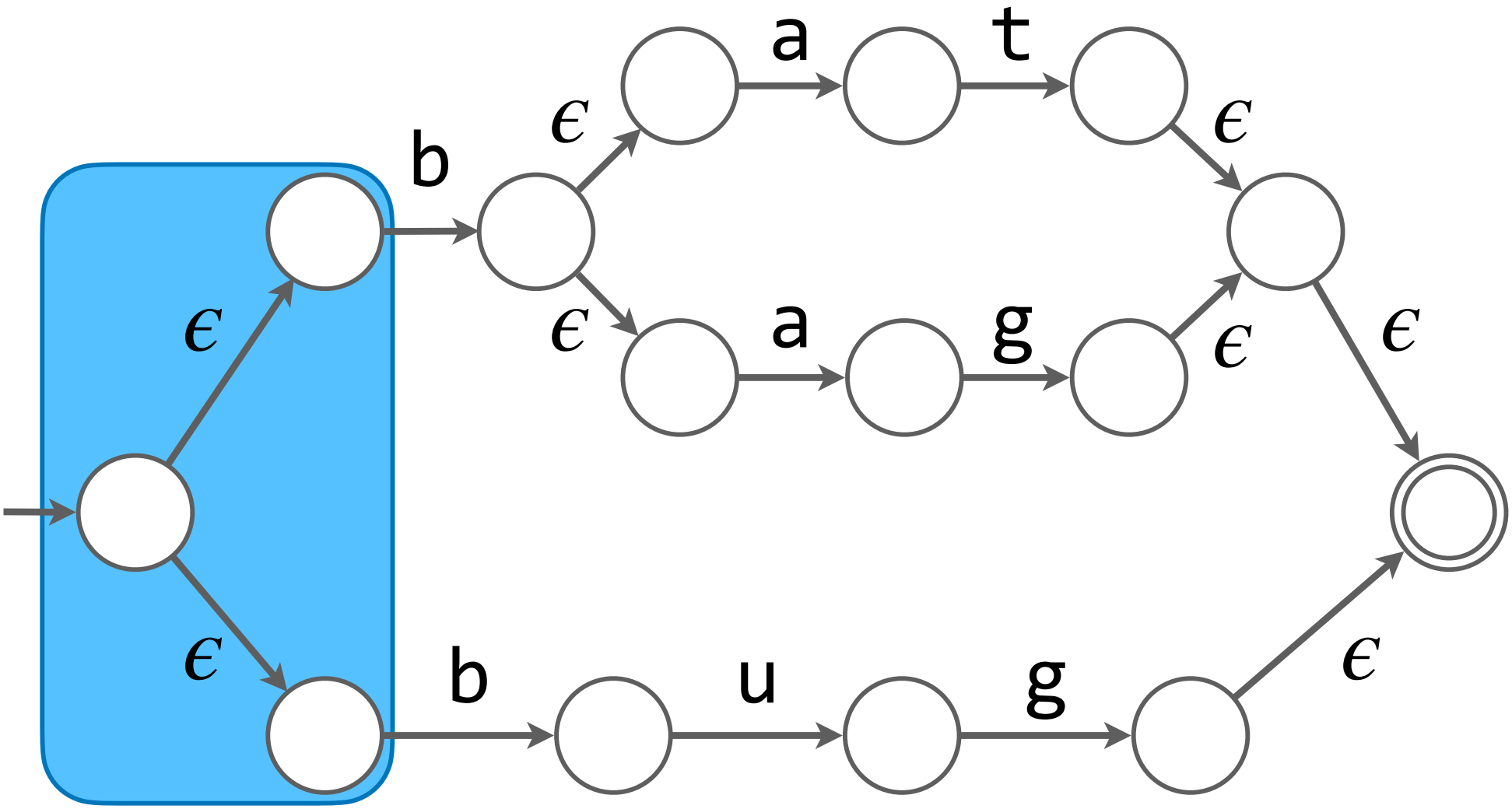
Exercise: (NFA → DFA)

build DFA for b(at|ag)|bug given the NFA



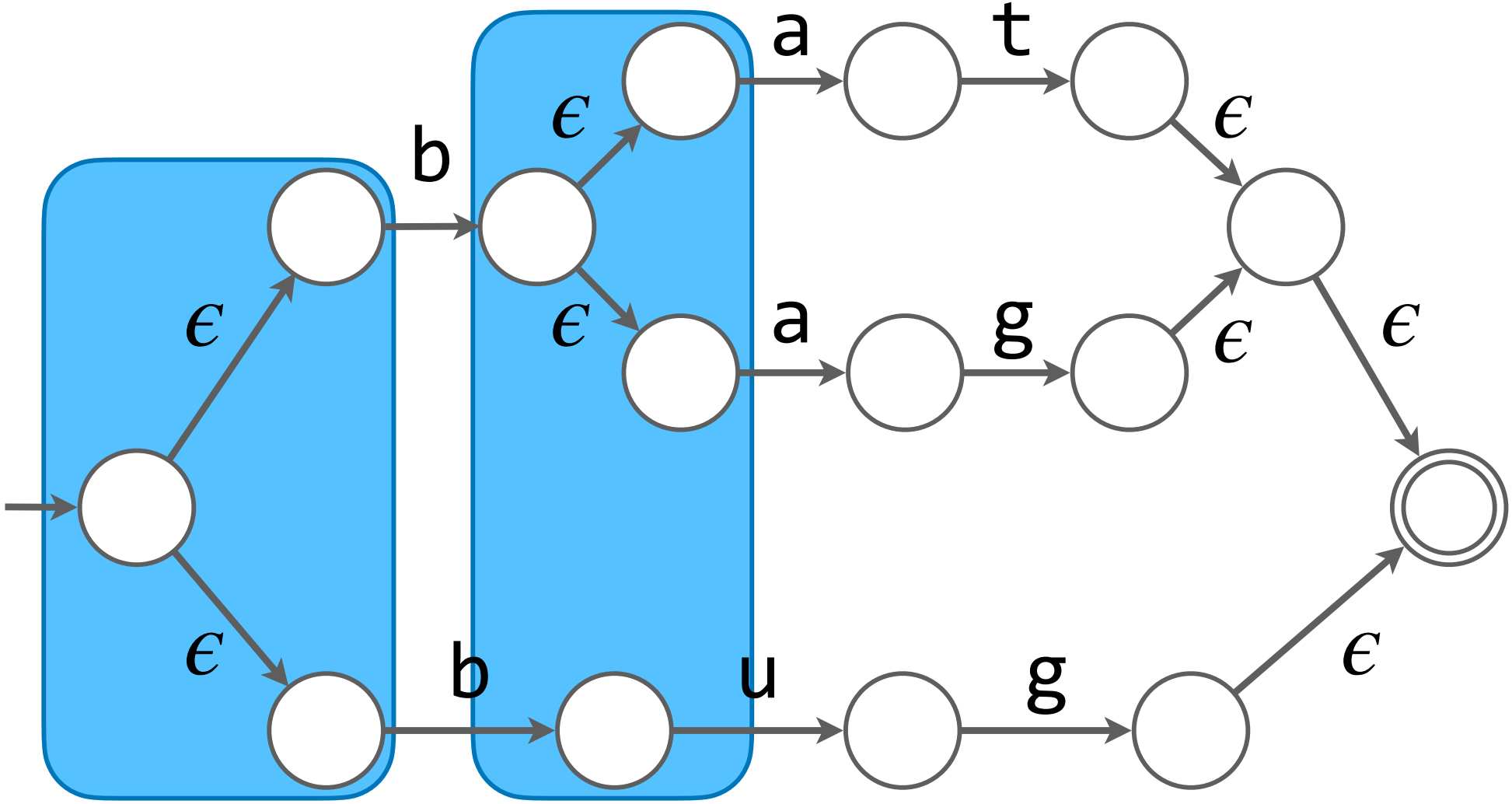
Exercise: (NFA → DFA)

build DFA for b(at|ag)|bug given the NFA



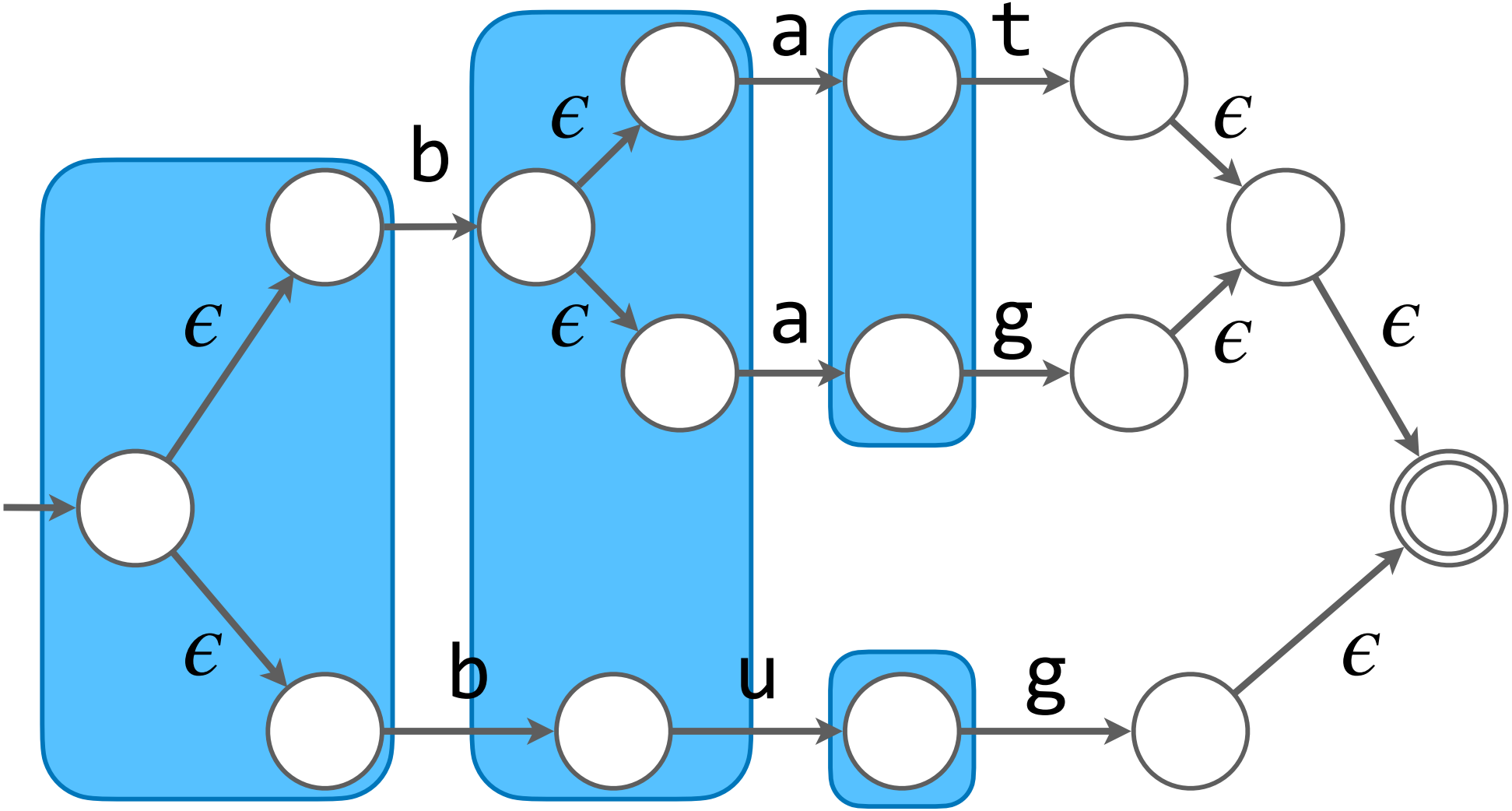
Exercise: (NFA → DFA)

build DFA for b(at|ag)|bug given the NFA



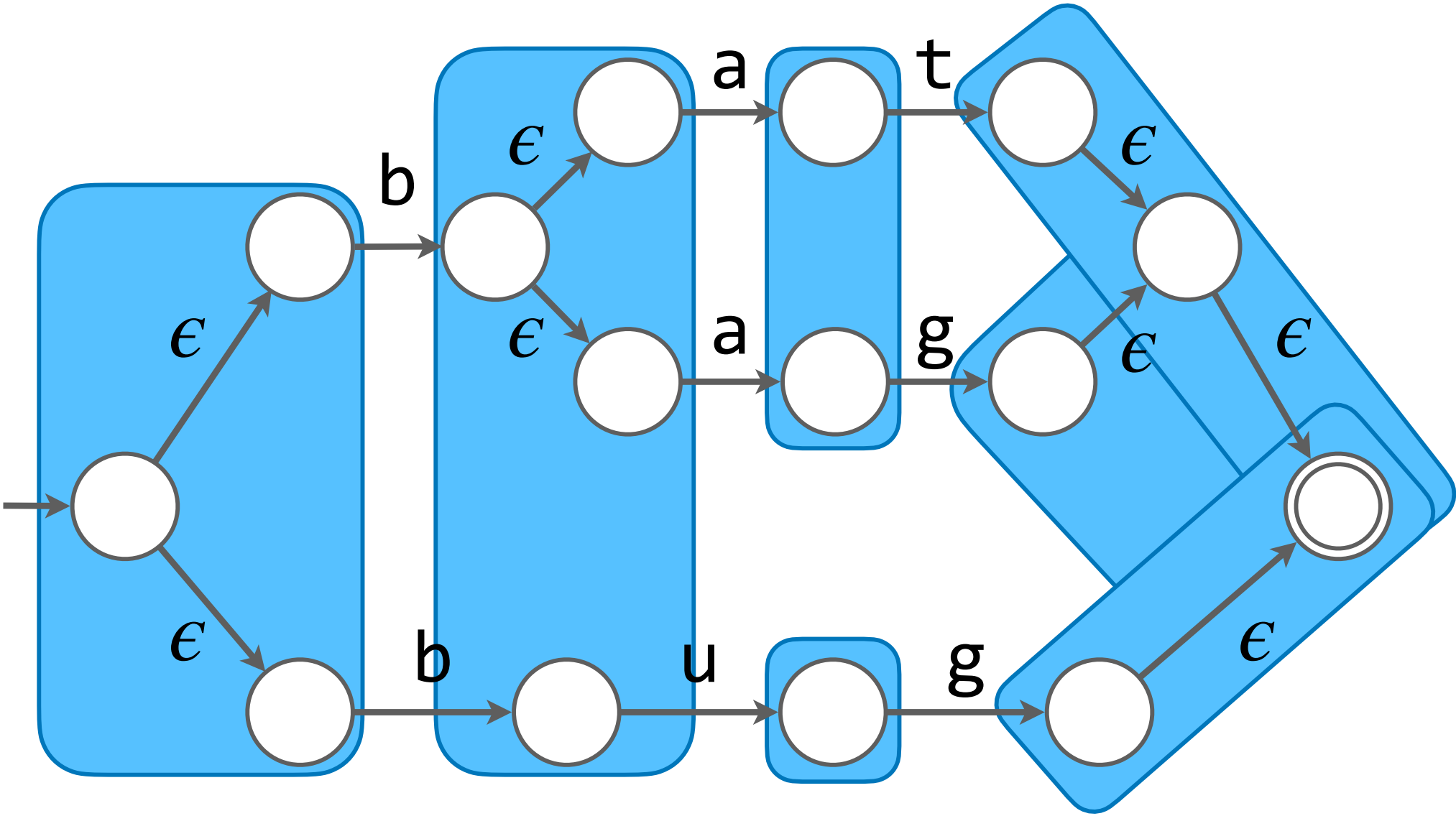
Exercise: (NFA → DFA)

build DFA for b(at|ag)|bug given the NFA



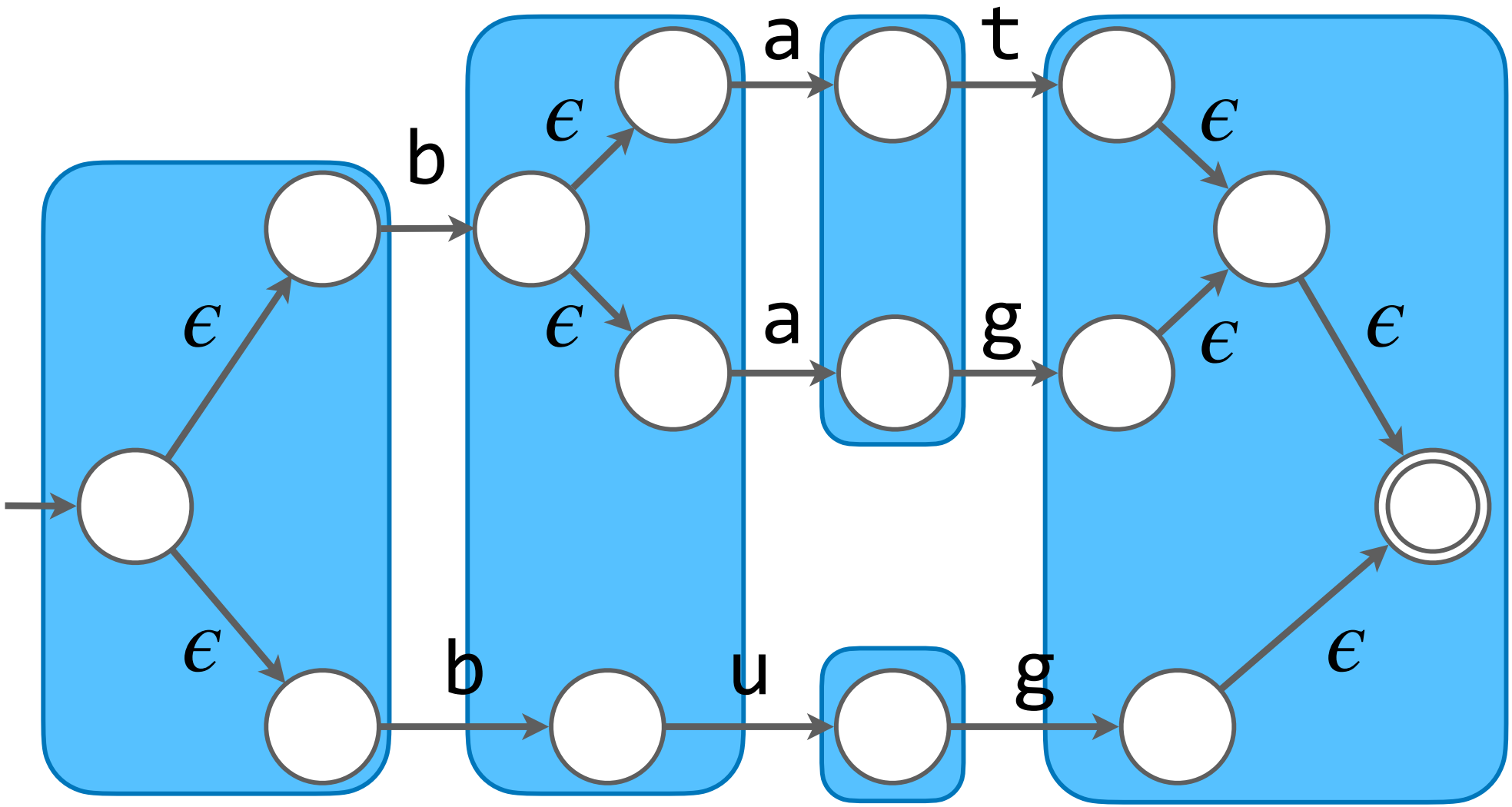
Exercise: (NFA → DFA)

build DFA for b(at|ag)|bug given the NFA



Exercise: (NFA → DFA)

build DFA for b(at|ag)|bug given the NFA



Outline

Review of Formal Languages, Grammars

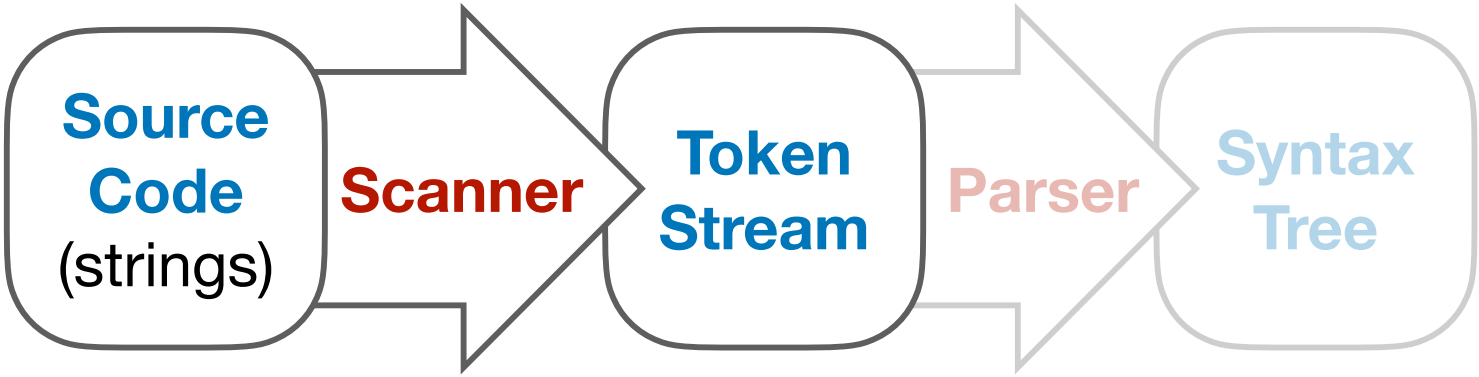
Lexical Specification of Prog. Lang.

Regular Expressions

Finite Automata — Recognize Reg. Exp.

Scanners & Tokens

Reminder: Scanners



- Input text

```
// this statement does very little
if (x >= y) y = 42;
```

- Token Stream

IF | LPAREN | ID(x) | GEQ | ID(y)

RPAREN | ID(y) | BECOMES | INT(42) | SCOLON

To Tokens

- A **scanner** is a DFA that finds the next token each time it's called, starting wherever it left off after the last token
- Every ***final*** state of a scanner DFA emits (returns) a token
- **Tokens** are units of scanner output (aka. words, lexemes)

`==`	becomes	EQUAL
`(`	becomes	LPAREN
`while`	becomes	LPAREN
`xyzzy`	becomes	ID(xyzzy)

- You choose the names for tokens
- Also, there may be additional data … \r\n might count lines; token data structure might include source line numbers

DFA → Scanner Code

- A couple of options
  1. Implement by hand
     a. Write one procedure for each state
     b. Write one procedure for all states/inputs
  2. Use tool to generate a table-driven scanner
     a. generate table data structure and drive using code that is parametric over the table
     b. generate code that has the table structure directly embedded in the code

DFA → Scanner Code (by hand)

a. implement by hand using procedures

- ✦ one procedure for each DFA state
- ✦ each procedure reads a character, and branches on it (using an if or switch statement)
- ✦ best with tail-call optimization or goto statements

- Pros
  - ✦ straightforward to write
  - ✦ reasonably *fast* at *compilation time*
- Cons
  - ✦ hand-writing scanner is a lot of tedious work
  - ✦ may diverge from the lexical specification

Notes on time ⏳ w/Compilers

- What do you mean by *fast*?
- Many different ***stages*** of time
- Key stages: from most frequent to least frequent
  - **Execution Time** Time to execute a compiled program
    - **Compilation Time** Time to execute the compiler
      - Time to build/compile the compiler itself
        - Programmer time to write/develop the compiler
- All are important, but not equally so
- Time consumed in more frequent stages is more important

DFA → Scanner Code (by hand)

b. implement by hand using a single procedure with multiple return points

- ✦ reads (potentially) multiple input characters (and may "look ahead")
- ✦ choices implemented with if, switch, loop control flow

- Pros
  - ✦ also straightforward to write
  - ✦ *faster* at *compilation time*
- Cons
  - ✦ still a lot of tedious work
  - ✦ still may diverge from the lexical specification

DFA → Scanner Code (generated)

a. use tool to generate a table-driven scanner

- ✦ one row of table for each state of DFA
- ✦ one column for each input character
- ✦ entry in table is action to take
  - next state to go to, *or* error, *or* accept + token + goto start

- Pros
  - ✦ more concise to specify
  - ✦ easier to ensure agreement with lexical specification
- Cons
  - ✦ "magic"

DFA → Scanner Code (generated)

b. use tool to generate a scanner *program*

- ✦ transitions embedded in code, so no table lookup
- ✦ choices use conditional statements, loops

- Pros
  - ✦ still more concise to specify
  - ✦ still more agreement with lexical specification
- Cons
  - ✦ still “magic”
- Potentially faster; depends on processor; e.g. code vs. data cache usage tradeoffs

Example: Handwritten Scanner

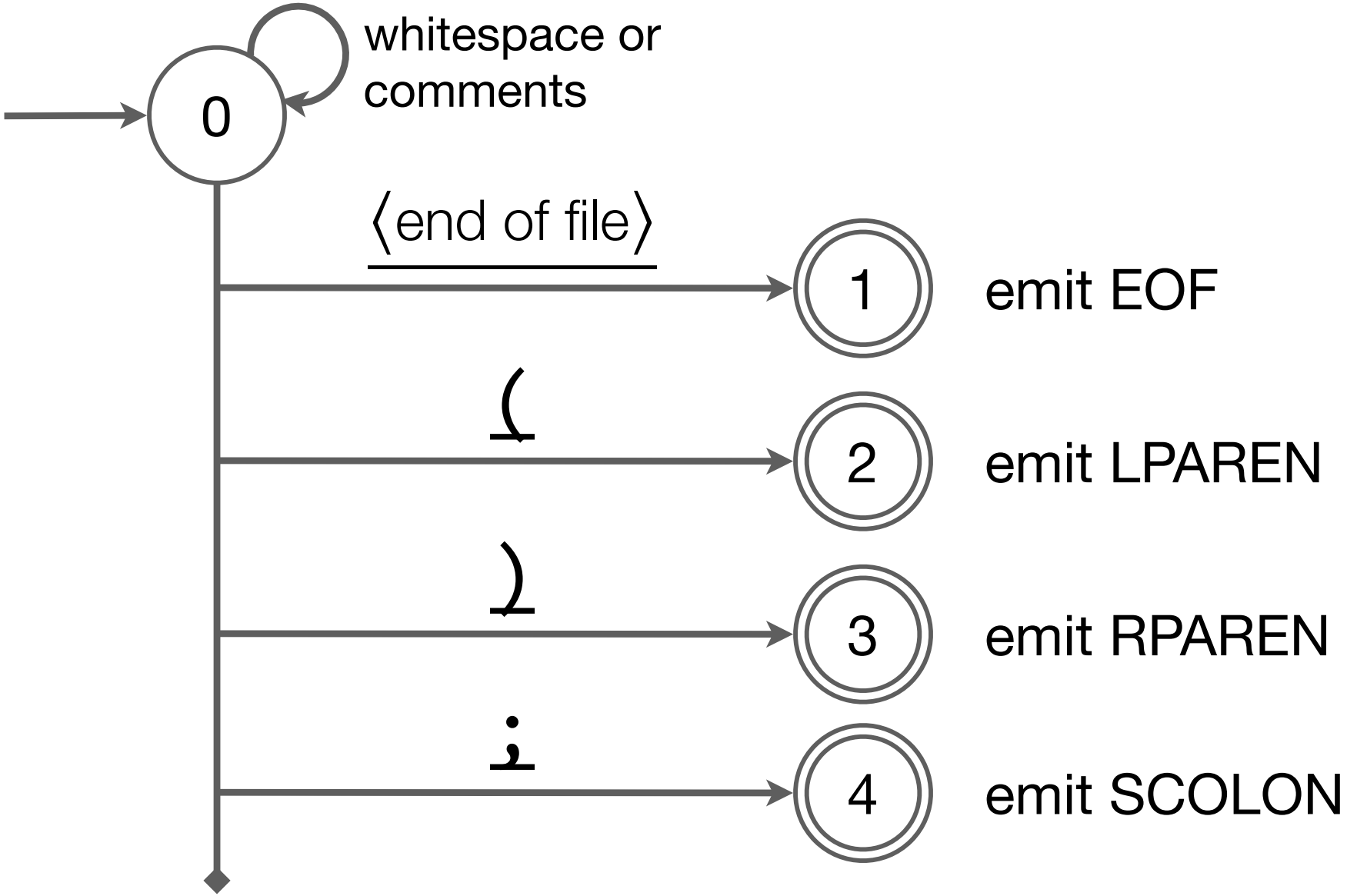
- **Specification** — Regular expression for each token

$$\begin{aligned}
\text{LPAREN} &::= \underline{(} & \text{EOF} &::= \underline{\langle \text{end of file} \rangle} \\
\text{RPAREN} &::= \underline{)} & \text{LESS} &::= \underline{<} \\
\text{SCOLON} &::= \underline{;} & \text{LEQ} &::= \underline{<=} \\
\text{NOT} &::= \underline{!} & \text{INT} &::= [\underline{0} - \underline{9}]^+ \\
\text{NEQ} &::= \underline{!=} & \text{ID} &::= [\underline{a}\text{-}\underline{zA}\text{-}\underline{Z}][\underline{a}\text{-}\underline{zA}\text{-}\underline{Z0}\text{-}\underline{9_}]^*
\end{aligned}$$

- Must merge the DFAs for all expressions into one DFA with labeled "final" states; at least one for each token
- Whitespace and errors handled as special cases
- Disclaimer: We will do a scanner generator for the project

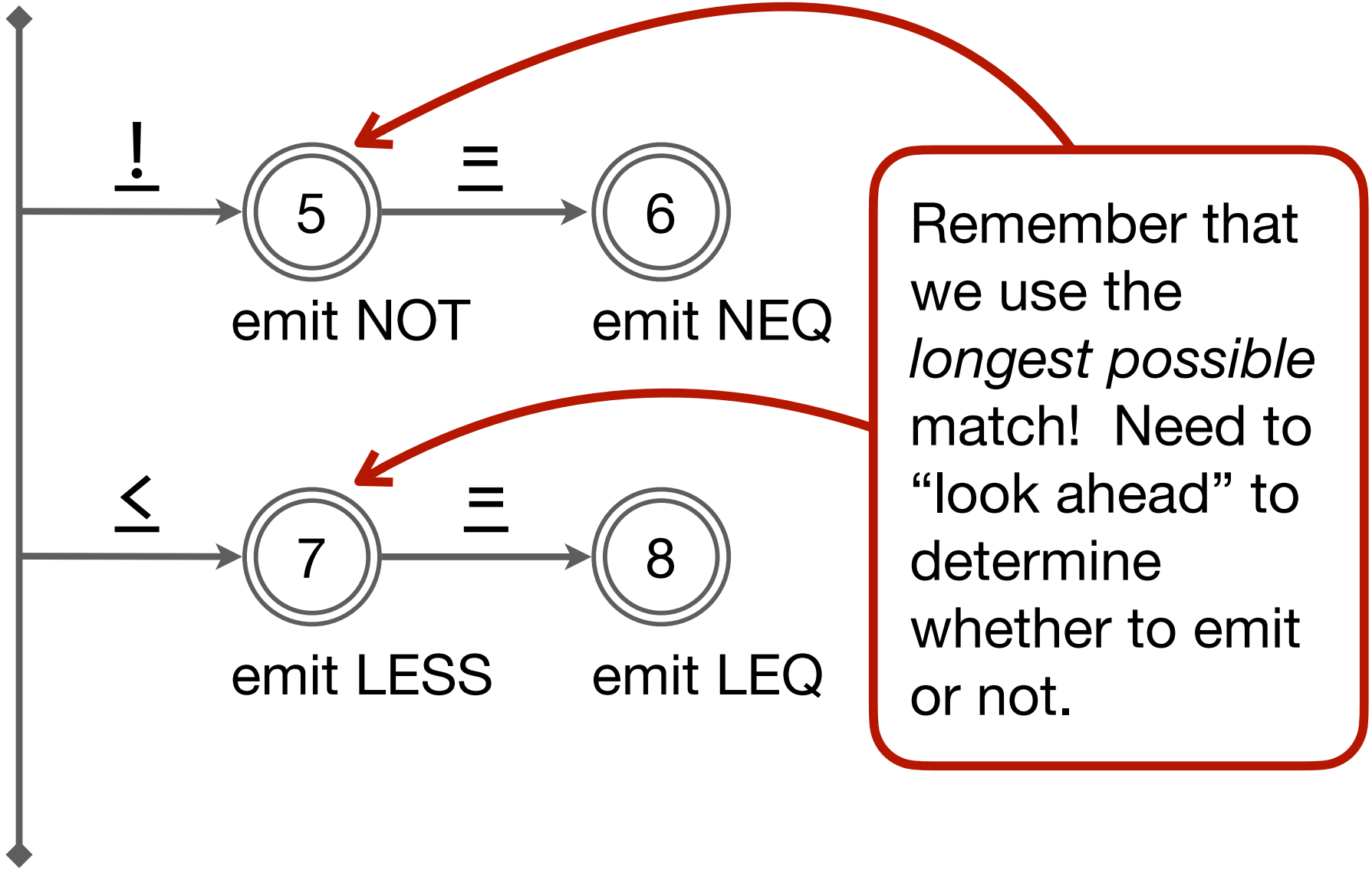
Example: Scanner DFA

1) Single Character Tokens



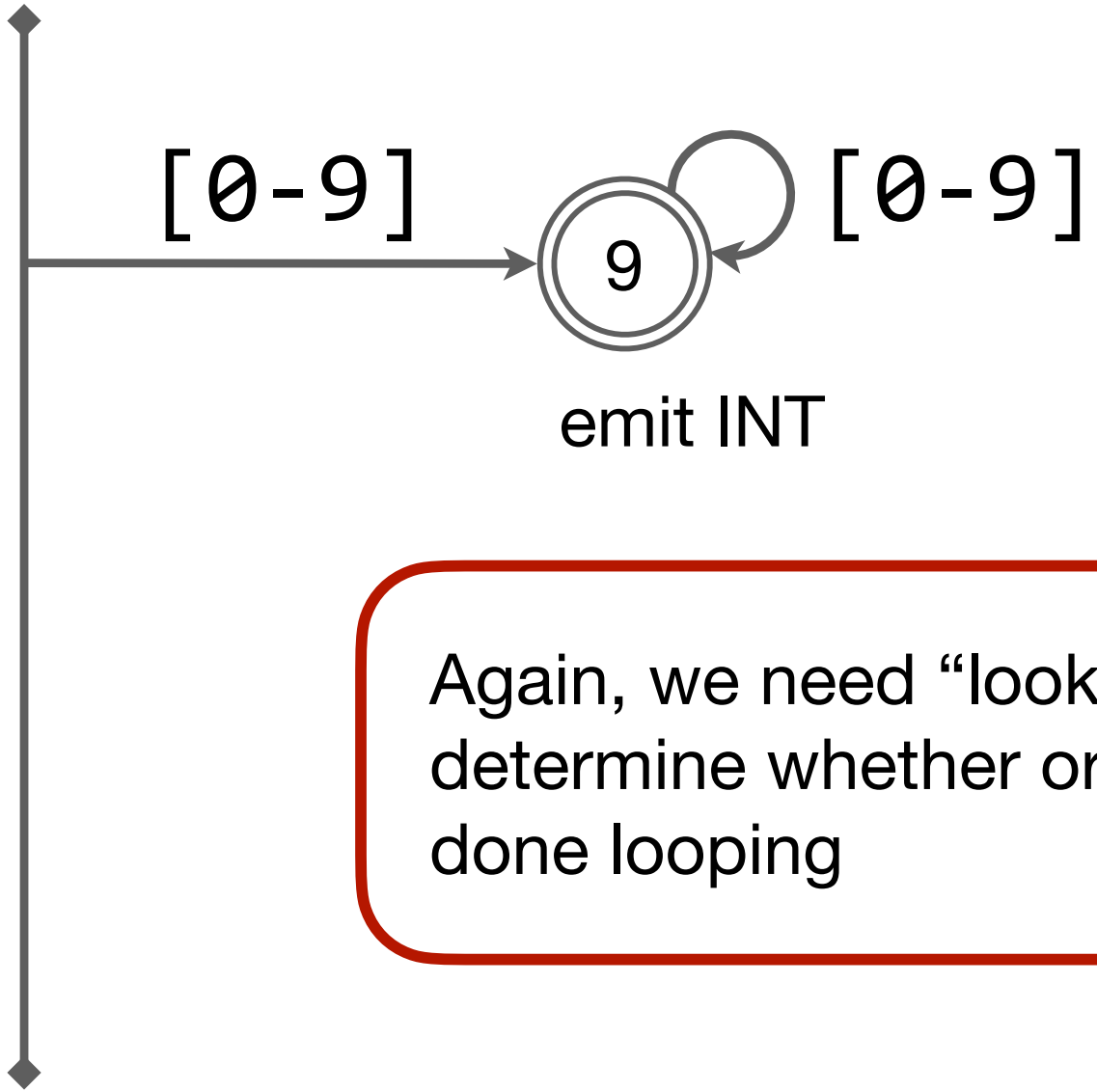
Example: Scanner DFA

2) Prefix ambiguity between tokens



Example: Scanner DFA

3) Numeric Literals



Again, we need “look ahead” to determine whether or not we’re done looping

Example: Scanner DFA

4) Identifiers & Keywords

Also needs look ahead

emit ID or keyword

- Strategies for disambiguating identifiers vs. keywords
  - ✦ Hand-written — before emitting an identifier, look it up in a keyword table (classic app. of perfect hashing)
  - ✦ Generated — Let generator create a DFA with lots of extra states (no lookup table step required)

Scanner: Hand Implementation

1) Token Representation

```
public class Token {
    public Kind kind;       // which "kind" of token?
    public int intVal;      // defined if kind == INT
    public String id;       // defined if kind == ID

    public int line;        // debug information

    public enum Kind {
        EOF,  LPAREN, RPAREN, NOT, NEQ,
        SCOLON, LESS, LEQ, INT, ID,
        // etc.
    }
}
```

Scanner: Hand Implementation

2) Scanner Helper Methods

```
public class Scanner {
    …
    public Scanner(String file_contents) { … }

    // get the next input character without consuming it
    public char lookahead() { … }

    // get the next input character and advance in input
    public char getch() { … }

    // advance in input past any whitespace and comments
    public char skipWhitespace() { … }

    …
```

Scanner: Hand Implementation

3) Start getting a token and single character tokens

```
public Token getToken() {
  skipWhitespace();

  if( /* no more input */ )
    return new Token(Token.Kind.EOF);

  char ch = getch();
  switch (ch) {
    case '(':  return new Token(Token.Kind.LPAREN);
    case ')':  return new Token(Token.Kind.RPAREN);
    case ';':  return new Token(Token.Kind.SCOLON);

    …
```

Scanner: Hand Implementation

4) Prefix ambiguous tokens

```
case '!':
  if(lookahead() == '=')
    return new Token(Token.Kind.NEQ);
  else
    return new Token(Token.Kind.NOT);

case '<':
  if(lookahead() == '=')
    return new Token(Token.Kind.LEQ);
  else
    return new Token(Token.Kind.LESS);

…
```

Scanner: Hand Implementation

5) Numeric Literals

```
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case ‘9':
  String num = ch;
  while( isDigit(lookahead()) )
    num = num + getch();

  return new Token(Token.Kind.INT,
                   Integer.parseInt(num));

…
```

Scanner: Hand Implementation

6) Identifiers and Keywords

```
case 'a': … case 'z':
case 'A': … case ‘Z':
  String id = ch;
  while( isDigit(lookahead()) ||
         isLetter(lookahead()) ||
         lookahead() == '_' ) {
    id = id + getch();
  }

  if(/* id is a keyword */)
    return new Token(kwdToken(id));
  else
    return new Token(Token.Kind.ID, id);
```

MiniJava Scanner Generation

- We'll use the jflex tool to automatically create a scanner from a specification file
- We'll use the CUP tool to automatically create a parser from a specification file
- Token class definitions are shared by jflex and CUP. Lexical classes are listed in CUP's input file, which generates the token class definition.
- Details in this week's sections

Next Time…

- HW 1 due Thursday
- First part of compiler project released (along with starter code) on Thursday
  - ✦ Make sure you have partner info entered

- Next Topic: Grammars & Parsing
  - ✦ We'll do LR parsing first (since it's needed for the project) and then circle back to do LL parsing
  - ✦ Good time to start reading ahead into Chapter 3