بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ـ دانشکدهٔ مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۴۰۲۲)

کامپایلر

حسین فلسفین

تذکر مجدد: توسیع‌ها موجب افزایش قدرت نمی‌شوند.

The extensions are convenient, but none extend the descriptive power of regular expressions: Any set of strings that can be described with these abbreviations could also be described by just the basic set of operators.

تذکر دیگر: فواصل هم مورد شناسایی قرار می‌گیرند اما به پارسر گزارش داده نمی‌شوند

We assign the lexical analyzer the job of stripping out white-space, by recognizing the "token" ws *defined by:*

$$ws \rightarrow (\mathbf{blank}|\mathbf{tab}|\mathbf{newline})^{+}$$

Here, $\mathbf{blank}$, $\mathbf{tab}$, *and* $\mathbf{newline}$ *are abstract symbols that we use to express the ASCII characters of the same names. Token* ws *is different from the other tokens in that, when we recognize it, we do not return it to the parser, but rather restart the lexical analysis from the character that follows the whitespace.*

شیوهٔ کتاب اژدها برای افزایش خوانایی عبارات منظم

3.3.4 Regular Definitions

For notational convenience, we may wish to give names to certain regular expressions and use those names in subsequent expressions, as if the names were themselves symbols. If Σ is an alphabet of basic symbols, then a *regular definition* is a sequence of definitions of the form:

$$\begin{aligned} d_1 &\rightarrow r_1 \\ d_2 &\rightarrow r_2 \\ &\cdots \\ d_n &\rightarrow r_n \end{aligned}$$

where:

1. Each d_i is a new symbol, not in Σ and not the same as any other of the d's, and
2. Each r_i is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$.

By restricting r_i to Σ and the previously defined d's, we avoid recursive definitions, and we can construct a regular expression over Σ alone, for each r_i. We do so by first replacing uses of d_1 in r_2 (which cannot use any of the d's except for d_1), then replacing uses of d_1 and d_2 in r_3 by r_1 and (the substituted) r_2, and so on. Finally, in r_n we replace each d_i, for $i = 1, 2, \ldots, n-1$, by the substituted version of r_i, each of which has only symbols of Σ.

Example 3.5 : C identifiers are strings of letters, digits, and underscores. Here is a regular definition for the language of C identifiers. We shall conventionally use italics for the symbols defined in regular definitions.

$$
\begin{array}{rcl}
\mathit{letter\_} & \rightarrow & \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \mid \texttt{\_} \\
\mathit{digit} & \rightarrow & \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
\mathit{id} & \rightarrow & \mathit{letter\_} \; (\; \mathit{letter\_} \mid \mathit{digit} \;)^*
\end{array}
$$

□

Example 3.6 : Unsigned numbers (integer or floating point) are strings such as `5280`, `0.01234`, `6.336E4`, or `1.89E-4`. The regular definition

$$
\begin{array}{rcl}
\mathit{digit} & \rightarrow & \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
\mathit{digits} & \rightarrow & \mathit{digit} \; \mathit{digit}^* \\
\mathit{optionalFraction} & \rightarrow & \texttt{.} \; \mathit{digits} \mid \epsilon \\
\mathit{optionalExponent} & \rightarrow & (\; \texttt{E} \; (\; \texttt{+} \mid \texttt{-} \mid \epsilon \;) \; \mathit{digits} \;) \mid \epsilon \\
\mathit{number} & \rightarrow & \mathit{digits} \; \mathit{optionalFraction} \; \mathit{optionalExponent}
\end{array}
$$

is a precise specification for this set of strings. That is, an *optionalFraction* is either a decimal point (dot) followed by one or more digits, or it is missing (the empty string). An *optionalExponent*, if not missing, is the letter E followed by an optional + or − sign, followed by one or more digits. Note that at least one digit must follow the dot, so *number* does not match `1.`, but does match `1.0`.

□

Example 3.7: Using these shorthands, we can rewrite the regular definition of Example 3.5 as:

$$
\begin{array}{rcl}
letter\_ & \rightarrow & \texttt{[A-Za-z\_]} \\
digit & \rightarrow & \texttt{[0-9]} \\
id & \rightarrow & letter\_ \; (\; letter\_ \mid digit \;)^{*}
\end{array}
$$

The regular definition of Example 3.6 can also be simplified:

$$
\begin{array}{rcl}
digit & \rightarrow & \texttt{[0-9]} \\
digits & \rightarrow & digit^{+} \\
number & \rightarrow & digits \; (\texttt{.} \; digits)? \; (\; \texttt{E} \; \texttt{[+-]}? \; digits \;)?
\end{array}
$$

□

دو قاعده که هنگام کار با ابزارهایی مانند Flex باید به آن توجه کرد

3.5.3 Conflict Resolution in Lex

We have alluded to the two rules that Lex uses to decide on the proper lexeme to select, when several prefixes of the input match one or more patterns:

1. Always prefer a longer prefix to a shorter prefix.
2. If the longest possible prefix matches two or more patterns, prefer the pattern listed first in the Lex program.

Example 3.12: The first rule tells us to continue reading letters and digits to find the longest prefix of these characters to group as an identifier. It also tells us to treat <= as a single lexeme, rather than selecting < as one lexeme and = as the next lexeme. The second rule makes keywords reserved, if we list the keywords before **id** in the program. For instance, if then is determined to be the longest prefix of the input that matches any pattern, and the pattern then precedes {id}, as it does in Fig. 3.23, then the token THEN is returned, rather than ID. □

دو قاعده که هنگام کار با ابزارهایی مانند Flex باید به آن توجه کرد

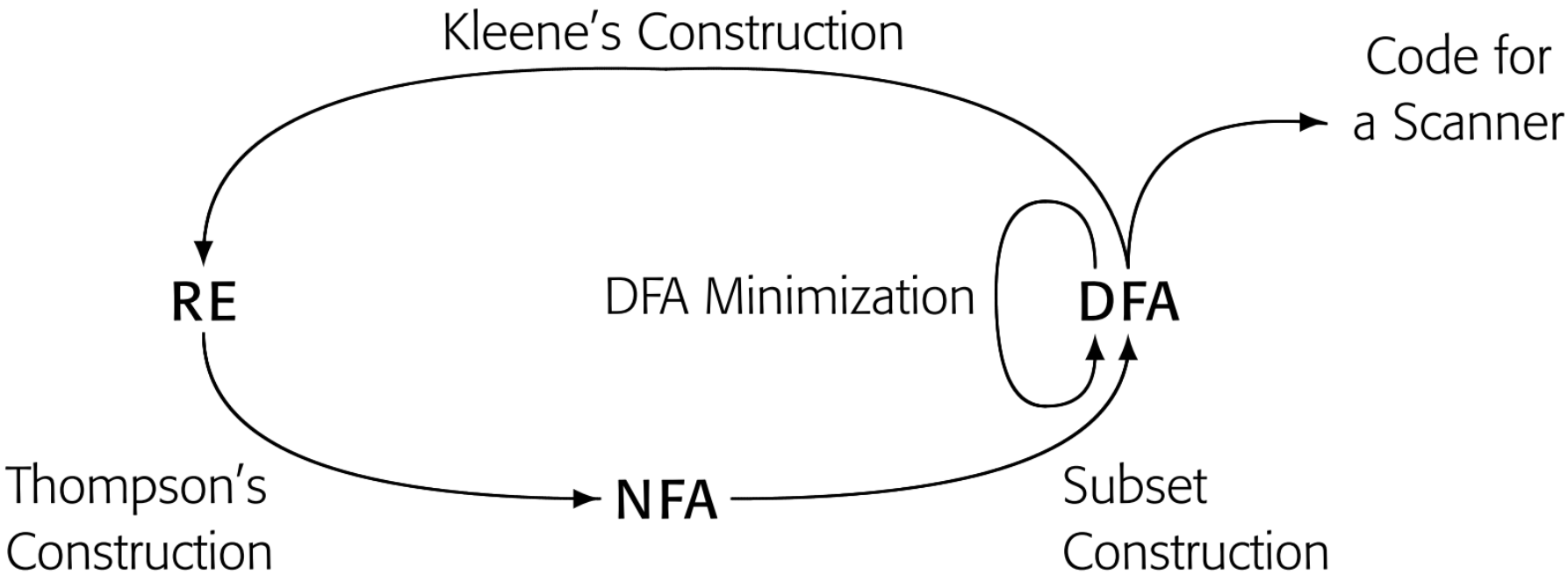
These rules are a bit ambiguous. For example, does `if8` match as a single identifier or as the two tokens `if` and `8`? Does the string `if 89` begin with an identifier or a reserved word? There are two important disambiguation rules used by Lex, JavaCC, SableCC, and other similar lexical-analyzer generators:

Longest match: The longest initial substring of the input that can match any regular expression is taken as the next token.

Rule priority: For a *particular* longest initial substring, the first regular expression that can match determines its token-type. This means that the order of writing down the regular-expression rules has significance.

Thus, `if8` matches as an identifier by the longest-match rule, and `if` matches as a reserved word by rule-priority.

The goal of our work with finite automata is to automate the derivation of scanners from a set of REs.



We learned how to express patterns using regular expressions. Now, we must study how to take the patterns for all the needed tokens and build a piece of code that examines the input string and finds a prefix that is a lexeme matching one of the patterns.

While the NFA is an abstract representation of an algorithm to recognize the strings of a certain language, the DFA is a simple, concrete algorithm for recognizing strings. It is fortunate indeed that every regular expression and every NFA can be converted to a DFA accepting the same language, because it is the DFA that we really implement or simulate when building lexical analyzers.

The regular expression is the notation of choice for describing lexical analyzers and other pattern-processing software. However, implementation of that software requires the simulation of a DFA or perhaps simulation of an NFA. Because an NFA often has a choice of move on an input symbol or on ε, or even a choice of making a transition on ε or on a real input symbol, its simulation is less straightforward than for a DFA. Thus often it is important to convert an NFA to a DFA that accepts the same language.

NFAs are a useful notion because it is easy to convert a (static, declarative) regular expression to a (simulatable, quasi-executable) NFA. Implementing deterministic finite automata (DFAs) as computer programs is easy. But implementing NFAs is a bit harder, since most computers don't have good "guessing" hardware. We can avoid the need to guess by trying every possibility at once (i.e., converting an NFA to a DFA).

https://en.wikipedia.org/wiki/Comparison_of_parser_generators

Name	Lexer algorithm	Output languages	Grammar, code	Development platform	License
Alex	DFA	Haskell	Mixed	All	Free, BSD
AnnoFlex	DFA	Java	Mixed	Java virtual machine	Free, BSD
Astir	DFA table driven, with branching	C++	Only grammar (actioned)	All	Free, MIT
AustenX	DFA	Java	Separate	All	Free, BSD
C# Flex	DFA	C#	Mixed	.NET CLR	Free, GNU GPL
C# Lex	DFA	C#	Mixed	.NET CLR	?
CookCC	DFA	Java	Mixed	Java virtual machine	Free, Apache 2.0
DFA	DFA compressed matrix	C, C++	Separate	Windows, Visual Studio	BSD
Dolphin	DFA	C++	Separate	All	Proprietary
Flex	DFA table driven	C, C++	Mixed	All	Free, BSD
gelex	DFA	Eiffel	Mixed	Eiffel	Free, MIT
golex	DFA	Go	Mixed	Go	Free, BSD-style
gplex	DFA	C#	Mixed	.NET CLR	Free, BSD-like
JFlex	DFA	Java	Mixed	Java virtual machine	Free, BSD
JLex	DFA	Java	Mixed	Java virtual machine	Free, BSD-like
lex	DFA	C	Mixed	POSIX	Partial, proprietary, CDDL
lexertl	DFA	C++	?	All	Free, GNU LGPL
Quex	DFA direct code	C, C++	Mixed	All	Free, GNU LGPL
Ragel	DFA	Go, C, C++, assembly	Mixed	All	Free, GNU GPL, MIT[1][2]
RE/flex	DFA direct code, DFA table driven, and NFA regex libraries	C++	Mixed	All	Free, BSD
re2c	DFA direct code	C, C++, Go, Rust	Mixed	All	Free, public domain

We shall now discover how `Lex` turns its input program into a lexical analyzer. At the heart of the transition is the formalism known as *finite automata.* These are essentially graphs, like transition diagrams, with a few differences:

1. Finite automata are *recognizers*; they simply say "yes" or "no" about each possible input string.

2. Finite automata come in two flavors:

   (a) *Nondeterministic finite automata* (NFA) have no restrictions on the labels of their edges. A symbol can label several edges out of the same state, and ϵ, the empty string, is a possible label.

   (b) *Deterministic finite automata* (DFA) have, for each state, and for each symbol of its input alphabet exactly one edge with that symbol leaving that state.

Both deterministic and nondeterministic finite automata are capable of recognizing the same languages. In fact these languages are exactly the same languages, called the *regular languages*, that regular expressions can describe.

تعریف کتاب اژدها از *NFA*

A *nondeterministic finite automaton* (NFA) consists of:

1. A finite set of states S.
2. A set of input symbols Σ, the *input alphabet*. We assume that ϵ, which stands for the empty string, is never a member of Σ.
3. A *transition function* that gives, for each state, and for each symbol in $\Sigma \cup \{\epsilon\}$ a set of *next states*.
4. A state s_0 from S that is distinguished as the *start state* (or *initial state*).
5. A set of states F, a subset of S, that is distinguished as the *accepting states* (or *final states*).

یک تعریف دیگر از *NFA* (کتاب مارتین)

A Nondeterministic Finite Automaton

A *nondeterministic finite automaton* (NFA) is a 5-tuple $(Q, \Sigma, q_0, A, \delta)$, where

Q is a finite set of states;
Σ is a finite input alphabet;
$q_0 \in Q$ is the initial state;
$A \subseteq Q$ is the set of accepting states;
$\delta : Q \times (\Sigma \cup \{\Lambda\}) \rightarrow 2^Q$ is the transition function.

For every element q of Q and every element σ of $\Sigma \cup \{\Lambda\}$, we interpret $\delta(q, \sigma)$ as the set of states to which the FA can move, if it is in state q and receives the input σ, or, if $\sigma = \Lambda$, the set of states other than q to which the NFA can move from state q without receiving any input symbol.

A deterministic finite automaton (DFA) is a special case of an NFA where:

1. *There are no moves on input ε, and*
2. *For each state s and input symbol a, there is exactly one edge out of s labeled a.*

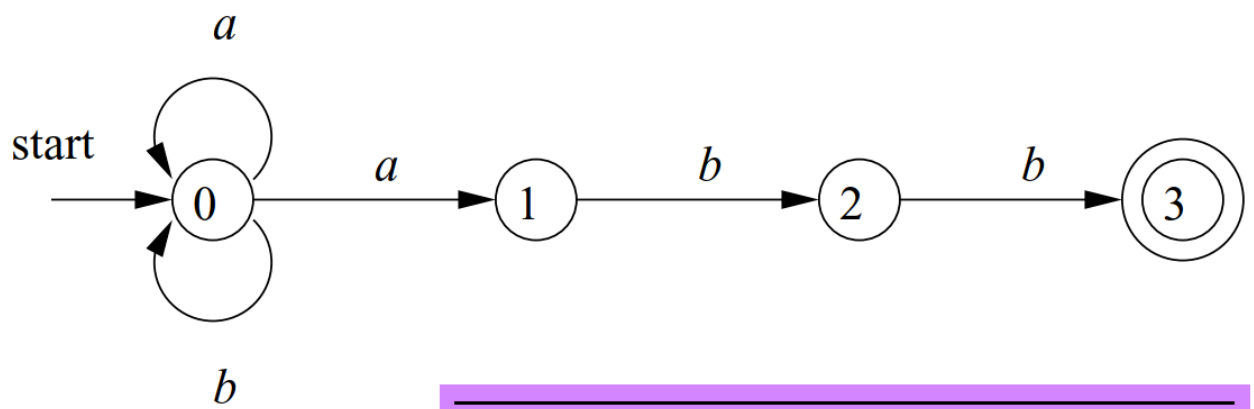
A Finite Automaton

A *finite automaton* (FA) is a 5-tuple $(Q, \Sigma, q_0, A, \delta)$, where

Q is a finite set of *states*;
Σ is a finite *input alphabet*;
$q_0 \in Q$ is the *initial* state;
$A \subseteq Q$ is the set of *accepting* states;
$\delta : Q \times \Sigma \rightarrow Q$ is the *transition* function.

For any element q of Q and any symbol $\sigma \in \Sigma$, we interpret $\delta(q, \sigma)$ as the state to which the FA moves, if it is in state q and receives the input σ.

The transition graph for an NFA recognizing the language of regular expression $(a|b)^*abb$



STATE	a	b	ϵ
0	$\{0,1\}$	$\{0\}$	$\emptyset$
1	$\emptyset$	$\{2\}$	$\emptyset$
2	$\emptyset$	$\{3\}$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$

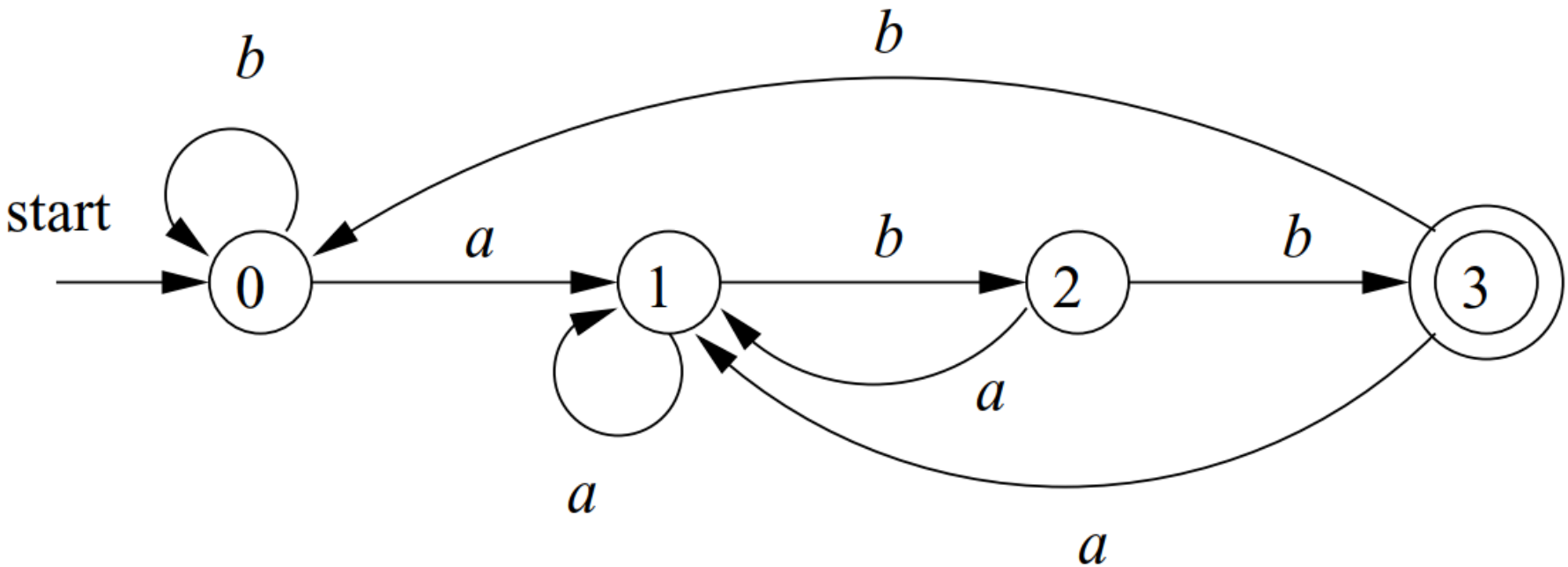


Figure 3.28: DFA accepting **(a|b)*abb**

Acceptance of Input Strings by Automata

An NFA accepts input string x if and only if there is some path in the transition graph from the start state to one of the accepting states, such that the symbols along the path spell out x. Note that ε labels along the path are effectively ignored, since the empty string does not contribute to the string constructed along the path.

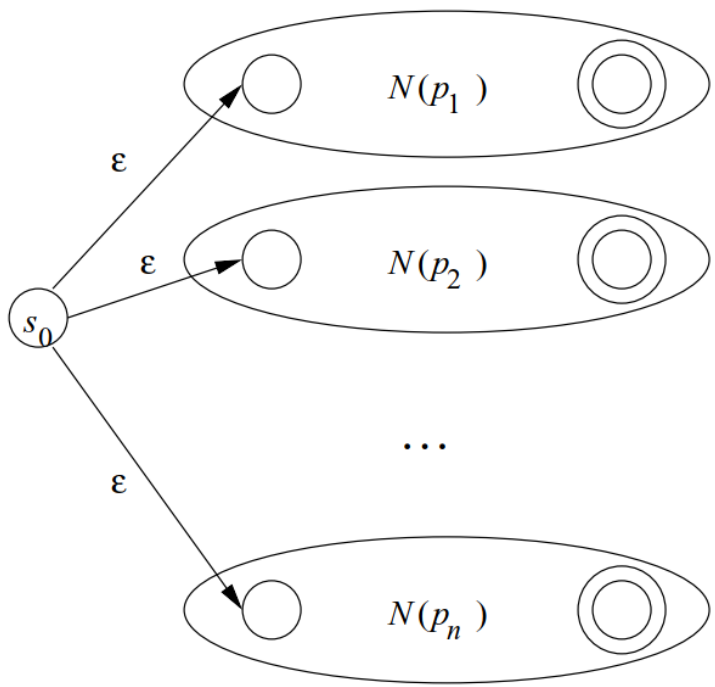
The formal definition of computation for an NFA is similar to that for a DFA. Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA and w a string over the alphabet Σ. Then we say that N ***accepts*** w if we can write w as $w = y_1 y_2 \cdots y_m$, where each y_i is a member of Σ_ε and a sequence of states $r_0, r_1, \ldots, r_m$ exists in Q with three conditions:

1. $r_0 = q_0$,
2. $r_{i+1} \in \delta(r_i, y_{i+1})$, for $i = 0, \ldots, m-1$, and
3. $r_m \in F$.

Condition 1 says that the machine starts out in the start state. Condition 2 says that state r_{i+1} is one of the allowable next states when N is in state r_i and reading y_{i+1}. Observe that $\delta(r_i, y_{i+1})$ is the *set* of allowable next states and so we say that r_{i+1} is a member of that set. Finally, condition 3 says that the machine accepts its input if the last state is an accept state.

مهم: راهی که برای به‌دست آوردن DFA باید طی شود

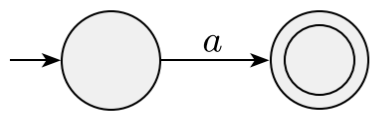
To construct the automaton, we begin by taking each regular expression pattern in the `Lex` *program and converting it to an NFA.* ***We need a single automaton*** *that will recognize lexemes matching any of the patterns in the program, so we* ***combine*** *all the NFA's into one by introducing a new start state with ε-transitions to each of the start states of the NFA's N_i for pattern p_i.*



Construction of an NFA from a Regular Expression
The McNaughton-Yamada-Thompson algorithm to convert an RE to an NFA.

Say that we have a regular expression R describing some language A, i.e., $L(R) = A$. We show how to convert R into an NFA recognizing A. Let's convert R into an NFA N. We consider the six cases in the formal definition of regular expressions.

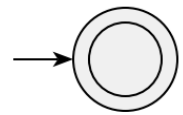
1. $R = a$ for some $a \in \Sigma$. Then $L(R) = \{a\}$, and the following NFA recognizes $L(R)$.



Note that this machine fits the definition of an NFA but not that of a DFA because it has some states with no exiting arrow for each possible input symbol. Of course, we could have presented an equivalent DFA here; but an NFA is all we need for now, and it is easier to describe.

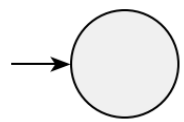
Formally, $N = (\{q_1, q_2\}, \Sigma, \delta, q_1, \{q_2\})$, where we describe δ by saying that $\delta(q_1, a) = \{q_2\}$ and that $\delta(r, b) = \emptyset$ for $r \neq q_1$ or $b \neq a$.

2. $R = \varepsilon$. Then $L(R) = \{\varepsilon\}$, and the following NFA recognizes $L(R)$.



Formally, $N = \big(\{q_1\}, \Sigma, \delta, q_1, \{q_1\}\big)$, where $\delta(r, b) = \emptyset$ for any r and b.

3. $R = \emptyset$. Then $L(R) = \emptyset$, and the following NFA recognizes $L(R)$.



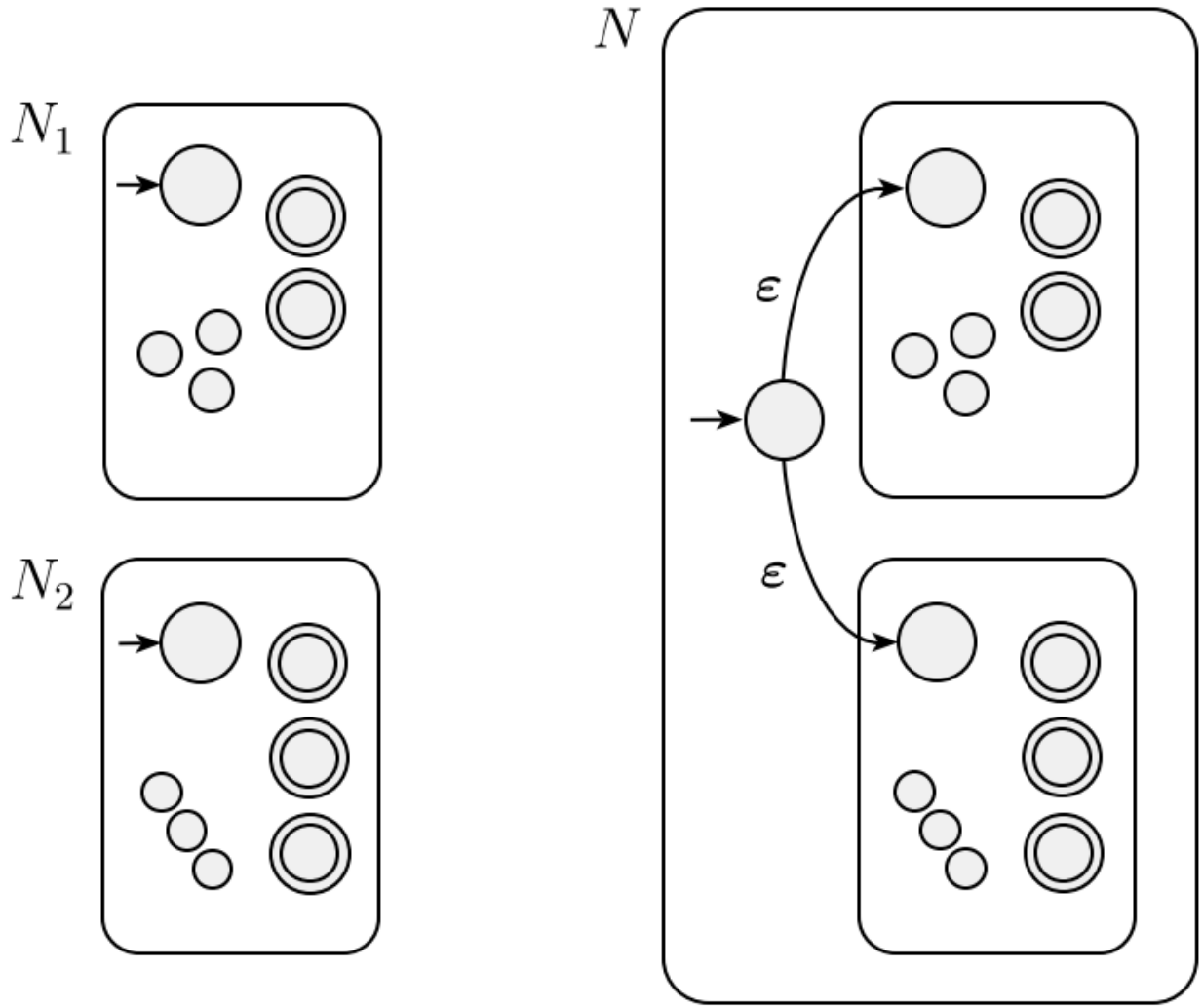
Formally, $N = \big(\{q\}, \Sigma, \delta, q, \emptyset\big)$, where $\delta(r, b) = \emptyset$ for any r and b.

4. $R = R_1 \cup R_2$.

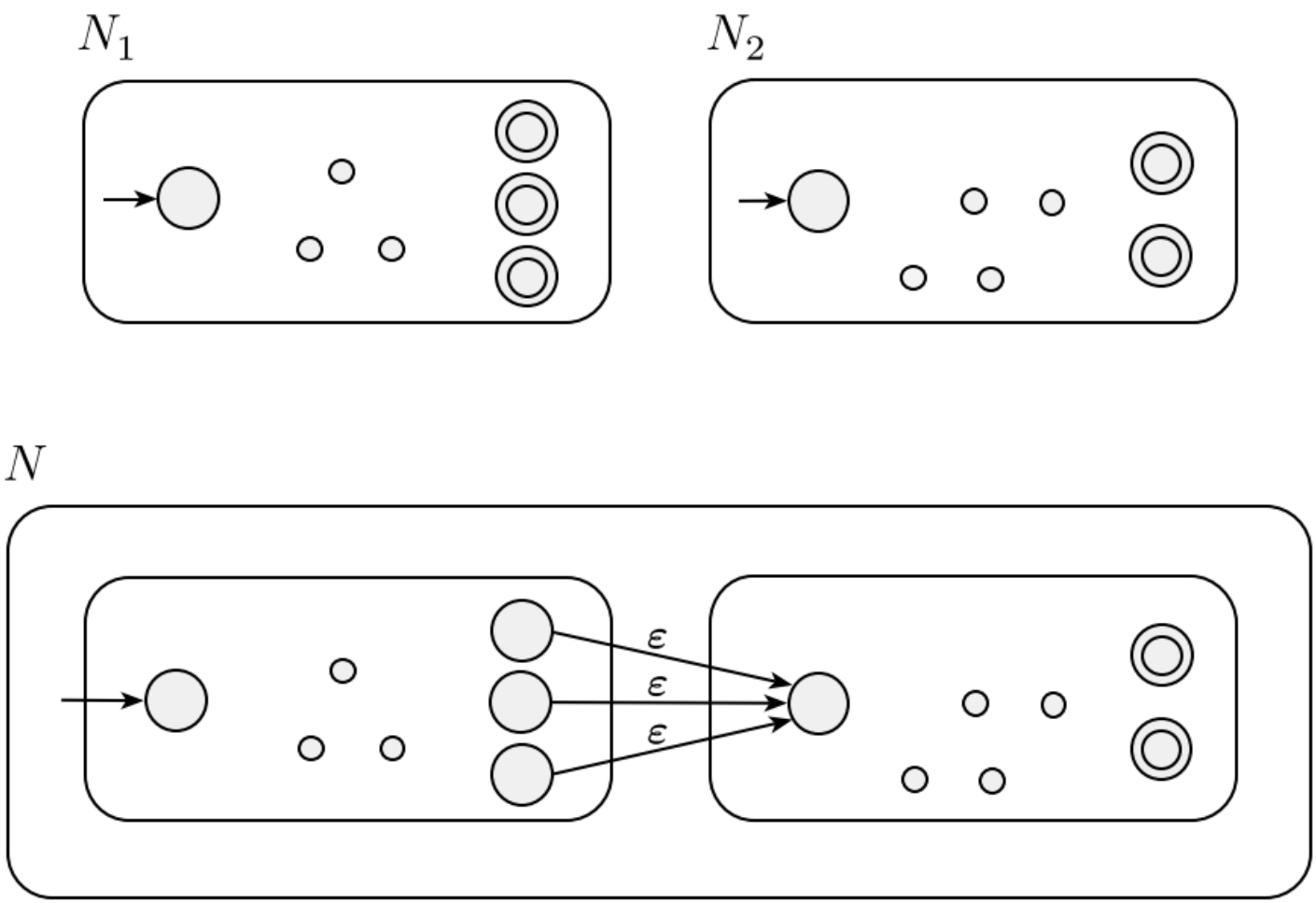
5. $R = R_1 \circ R_2$.

6. $R = R_1^*$.

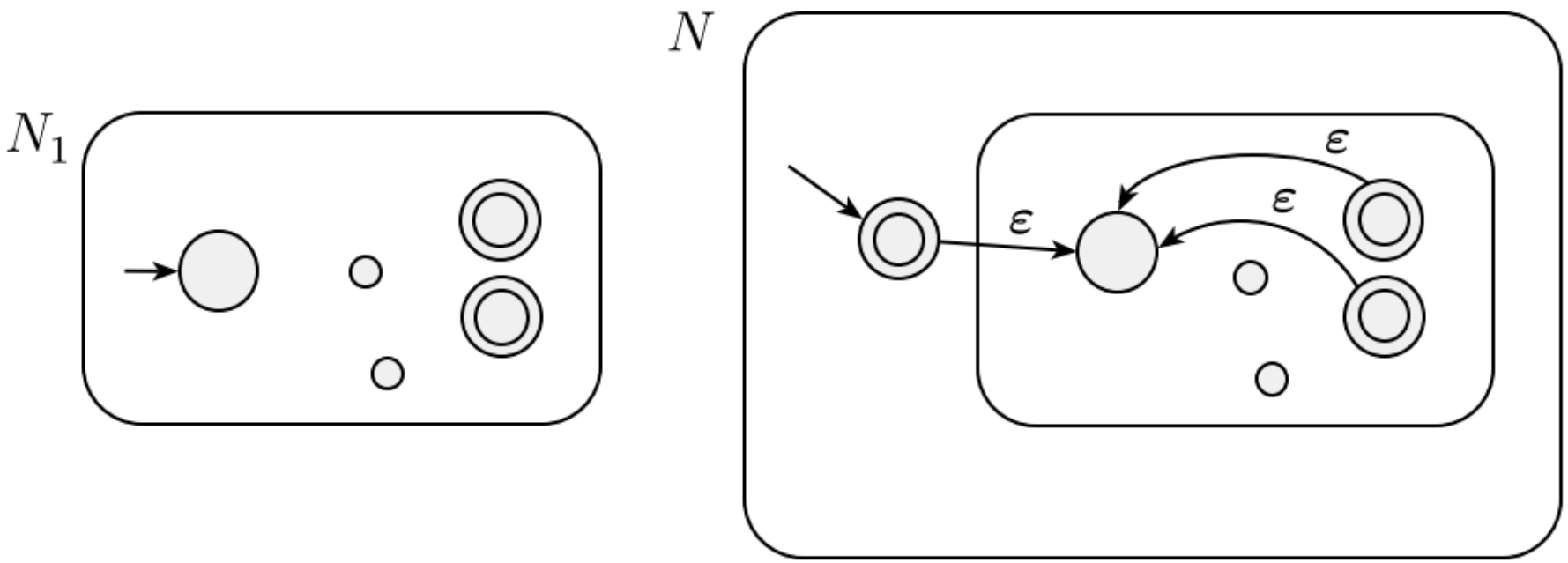
For the last three cases, we use the constructions given in the proofs that the class of regular languages is closed under the regular operations. In other words, we construct the NFA for R from the NFAs for R_1 and R_2 (or just R_1 in case 6) and the appropriate closure construction.



Construction of an NFA N to recognize $A_1 \cup A_2$

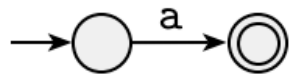


Construction of N to recognize $A_1 \circ A_2$

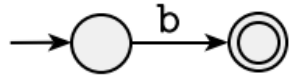


Construction of N to recognize A^*

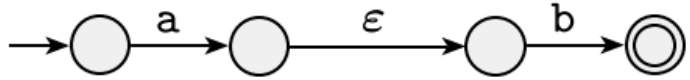
a



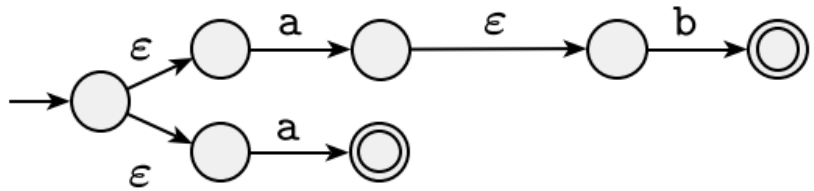
b



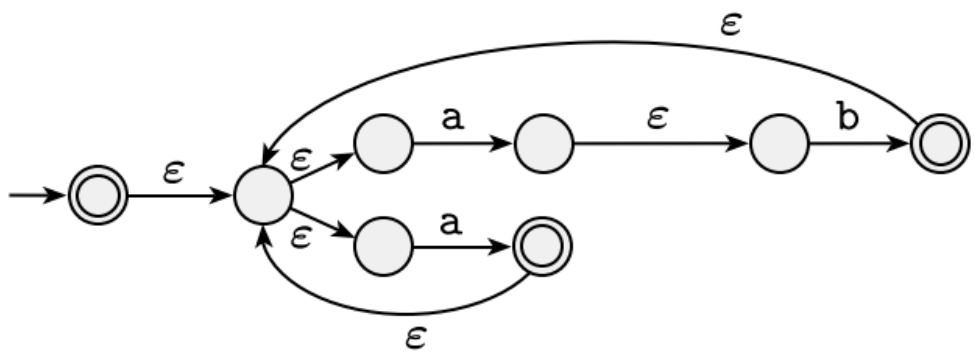
ab



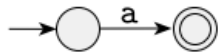
ab ∪ a



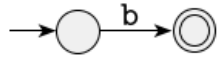
(ab ∪ a)*



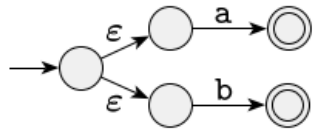
a



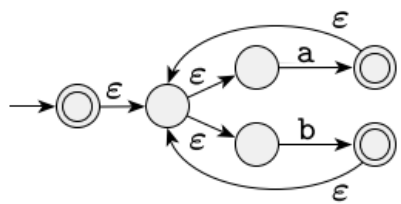
b



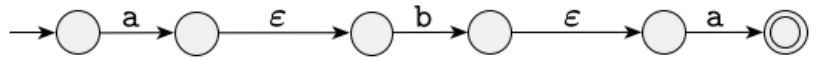
a∪b



(a∪b)*



aba



(a∪b)*aba