Lecture 4: Finite Automata

Administrivia

- Please select an open discussion section. If there is no room, please go to one where you can fit in the room. We may open another section, depending on demand.
- Please get a Unix instructional account (here) and a github account (at github.com).
- I'd like to have teams formed by Friday, if possible.

An Alternative Style for Describing Languages

- Rather than giving a single pattern, we can give a set of rules of the form:

$$A:\ \alpha_1\alpha_2\cdots\alpha_n,\ \ n \geq 0,$$

  where

  - A is a symbol that is intended to stand for a language (set of strings)—a *metavariable* or *nonterminal symbol.*
  - Each α_i is either a literal character (like `"a"`) or a nonterminal symbol.

- The interpretation of this rule is

  One way to form a string in $L(A)$ (the language denoted by A) is to concatenate one string each from $L(\alpha_1),\ L(\alpha_2),\ldots$.

  (where $L("c")$ is just the language $\{"c"\}$).

- This is *Backus-Naur Form (BNF)*. A set of rules is a *grammar*. One of the nonterminals is designated as its *start symbol* denoting the language described by the grammar.

- **Aside:** You'll see that ':' written many different ways, such as '::=', '$\longrightarrow$', etc.

Some Abbreviations

- The basic form from the last slide is good for formal analysis, but not for writing.
- So, we can allow some abbreviations that are obviously exandable into the basic forms:

Abbreviation	Meaning
$A:\ \mathcal{R}_1 \mid \cdots \mid \mathcal{R}_n$	$A:\ \mathcal{R}_1$ $\vdots$ $A:\ \mathcal{R}_n$
$A:\ \cdots(\mathcal{R})\cdots$	$B:\ \mathcal{R}$ $A: \cdots B \cdots$
$A:$ "c_1" | $\cdots$ | "c_n" (likewise other character classes)	$[c_1 \cdots c_n]$

Some Technicalities

- From the definition, each nonterminal in a grammar defines a language. Often, we are interested in just one of them (the *start symbol*), and the others are auxiliary definitions.
- The definition of what a rule means ("One way to form a string in $L(A)$ is...") leaves open the possibility that there are other ways to form items in $L(A)$ than covered in the rule.
- We need that freedom in order to allow multiple rules for A, but we don't really want to include strings that aren't covered by some rule.
- So precise mathematical definitions throw in sentences like:

  A grammar defines the *minimal* languages that contain all strings that satisfy the rules.

A Big Restriction (for now)

- For the time being, we'll also add a restriction. In each rule:

$$A :\ \alpha_1\alpha_2 \cdots \alpha_n, \quad n \geq 0,$$

we'll require that if α_i is a nonterminal symbol, then either

  - All the rules for that symbol have to occured before all the rules for A, or
  - $i = n$ (i.e., is the last item) and α_n is A.

- We call such a restricted grammar a *Type 3* or *regular* grammar. The languages definable by regular grammars are called *regular languages.*

Claim: Regular languages are exactly the ones that can be defined by regular expressions.

Proof of Claim (I)

- Start with a regular expression, $\mathcal{R}$, and make a (possibly not yet valid) rule,

  R: $\mathcal{R}$

- Create a new (preceding) rule for each parenthesized expression.
- This will leave just the constructs '$X*$', '$X+$', and '$X?$'. What do we do with them?

Proof of Claim (II)

Replace construct. . .	. . . with Q, where
$R*$	

Proof of Claim (II)

Replace construct. . .	. . . with Q, where
R*	`Q :` `Q : R Q`

R+

Proof of Claim (II)

Replace construct. . .	. . . with Q, where
$R*$	`Q :` `Q : R Q`
$R+$	`Q : R` `Q : R Q`

$R?$

Proof of Claim (II)

Replace construct...	**...with Q, where**
R*	`Q :` `Q : R Q`
R+	`Q : R` `Q : R Q`
R?	`Q :` `Q : R`

Example

- Consider the regular expression `("+"|"-")?("0"|"1")+`

1. `R: ("+"|"-")?("0"|"1")+` *replace with . . .*

2. Q_1: `"+" | "-"`
   Q_2: `"0" | "1"`
   R: Q_1? Q_2+ *replace with . . .*

3. Q_3: ϵ | Q_1
   Q_4: Q_2 | Q_2 Q_4
   R: Q_3 Q_4

Side Note: The Empty Language

- The grammar for the empty language is a bit non-intuitive:

```
Q: Q
```

- *Any* set of strings, `Q`, satisfies this rule.
- Hence, by the implicit rule that we choose the *smallest* solution that satisfies all rules, `Q` represents the empty set.

Classical Pattern-Matching Implementation

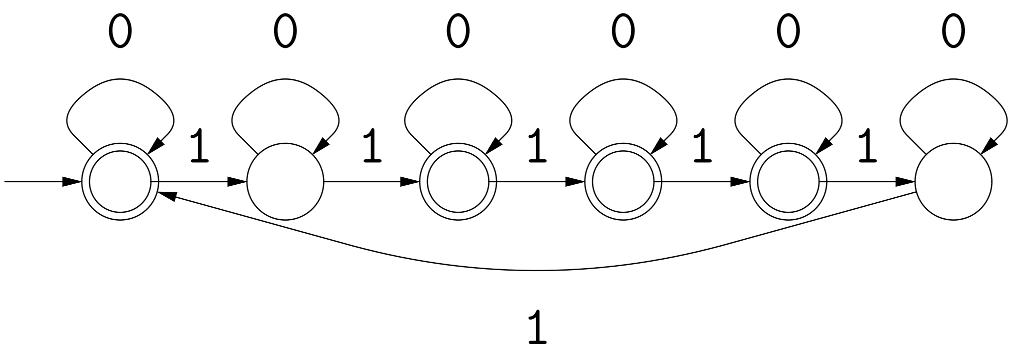
- For compilers, can generally make do with “classical” regular expressions.
- Implementable using *finite(-state) automata* or *FAs.* (“Finite state” = “finite memory”).
- Classical construction:

  regular expression $\Rightarrow$ nondeterministic FA (NFA)
  $\Rightarrow$ deterministic FA (DFA) $\Rightarrow$ table-driven program.

Review: FA operation

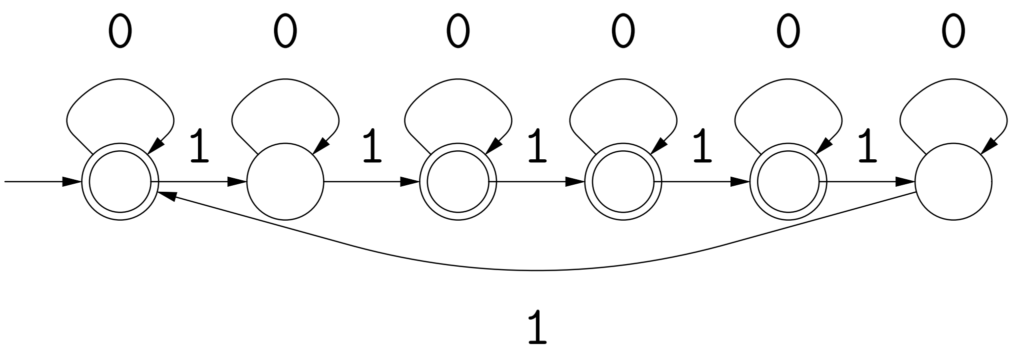
- A FA is a graph whose nodes are states (of memory) and whose edges are state transitions. There are a finite number of nodes.
- One state is the designated start state.
- Some subset of the nodes are final states.
- Each transition is labeled with a set of symbols (characters, etc.) or ϵ.
- A FA recognizes a string $c_1 c_2 \cdots c_n$ if there is a path (sequence of edges) from the start state to a final state such that the labels of the edges in sequence, aside from ϵ edges, respectively contain $c_1, c_2, \ldots, c_n$.
- If the edges leaving any node have disjoint sets of characters and if there are no ϵ nodes, FA is a DFA, else an NFA.

Example: What does this DFA recognize?



What is the simplest equivalent NFA you can think of?

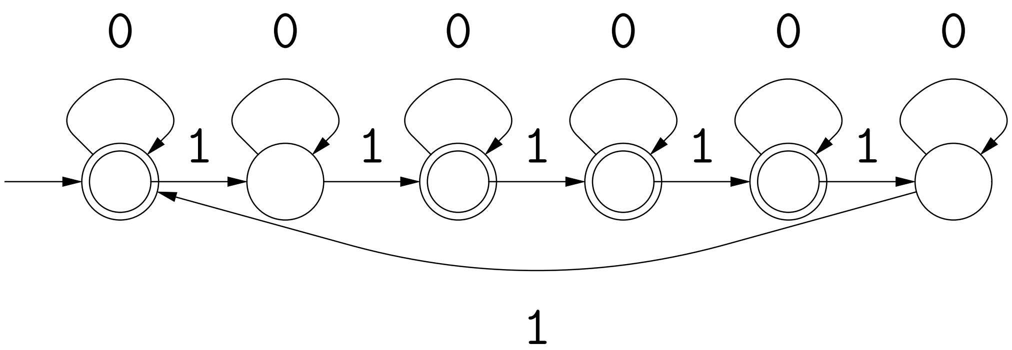
Example: What does this DFA recognize?



Bit strings with # of 1's divisible by 2 or 3.

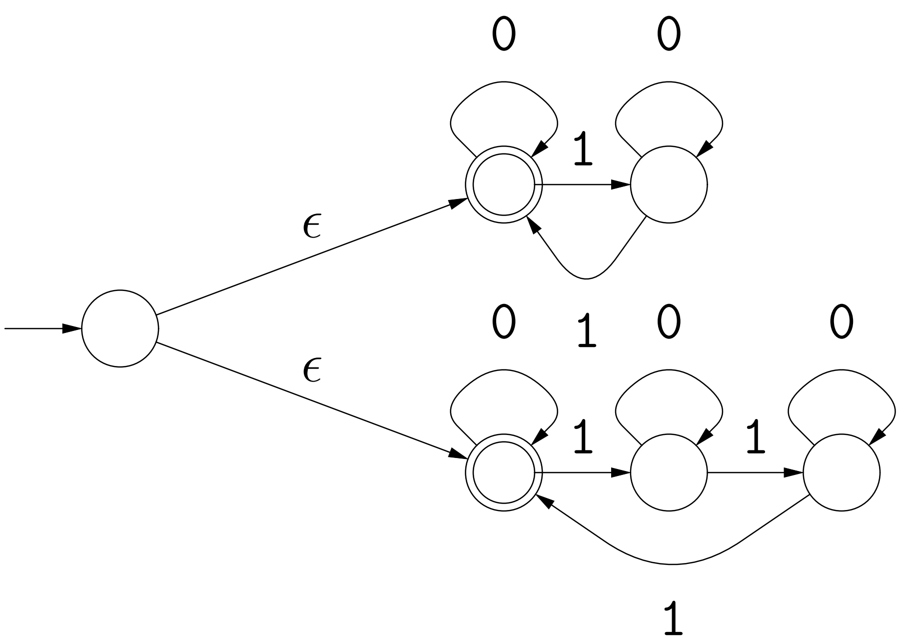
What is the simplest equivalent NFA you can think of?

Example: What does this DFA recognize?

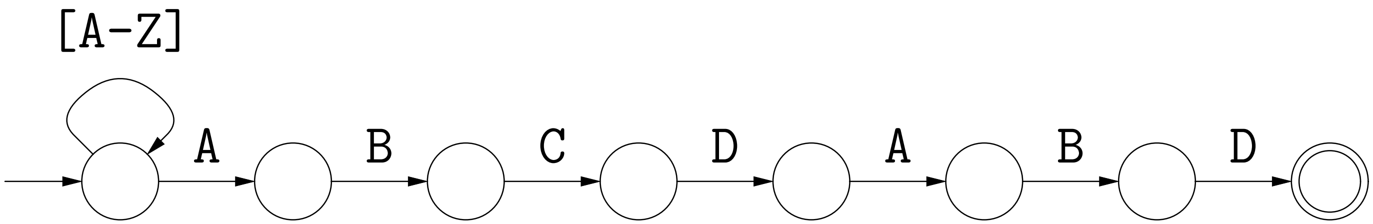


Bit strings with # of 1's divisible by 2 or 3.

What is the simplest equivalent NFA you can think of?

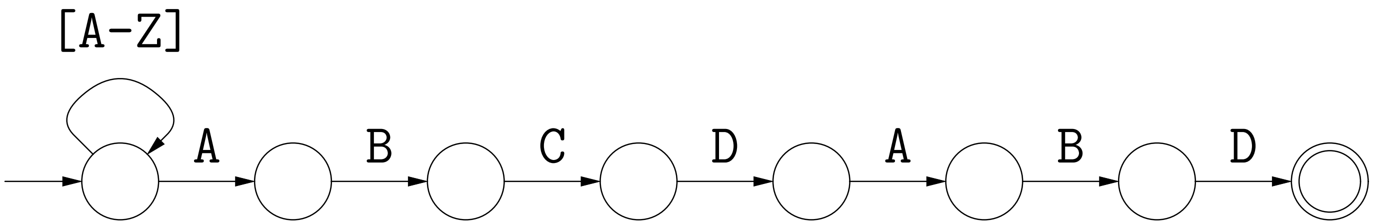


Example: What does this NFA recognize?



What is the simplest equivalent DFA you can think of?

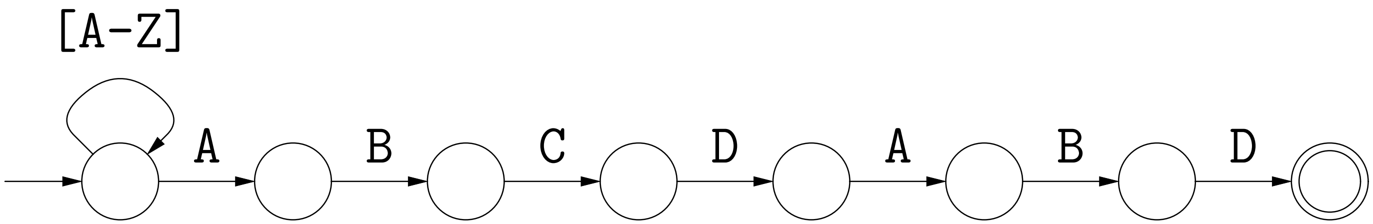
Example: What does this NFA recognize?



Strings of capitals ending in ABCDABD.

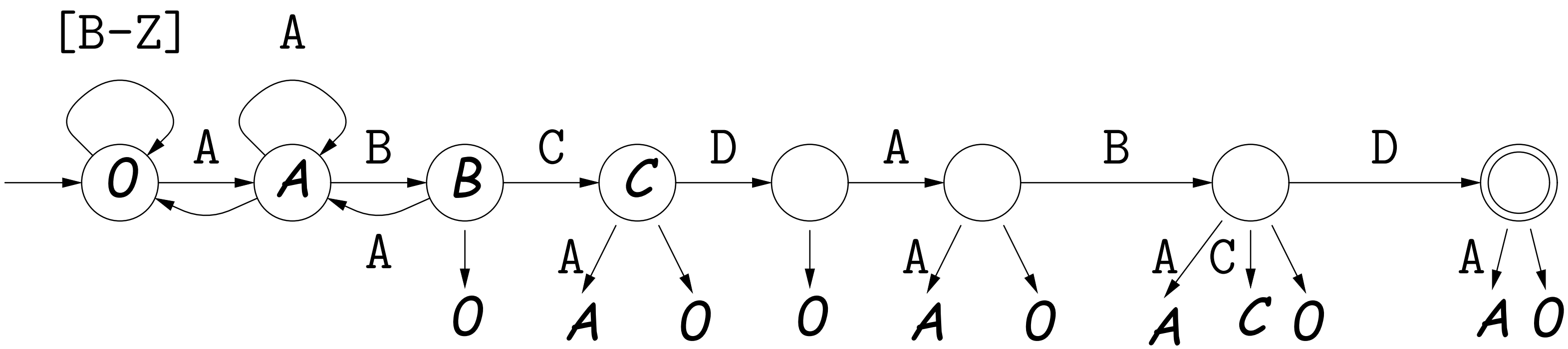
What is the simplest equivalent DFA you can think of?

Example: What does this NFA recognize?



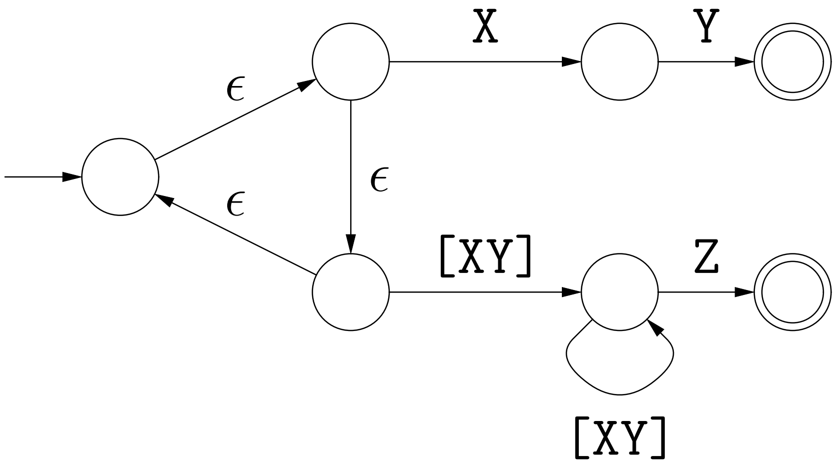
Strings of capitals ending in ABCDABD.

What is the simplest equivalent DFA you can think of?



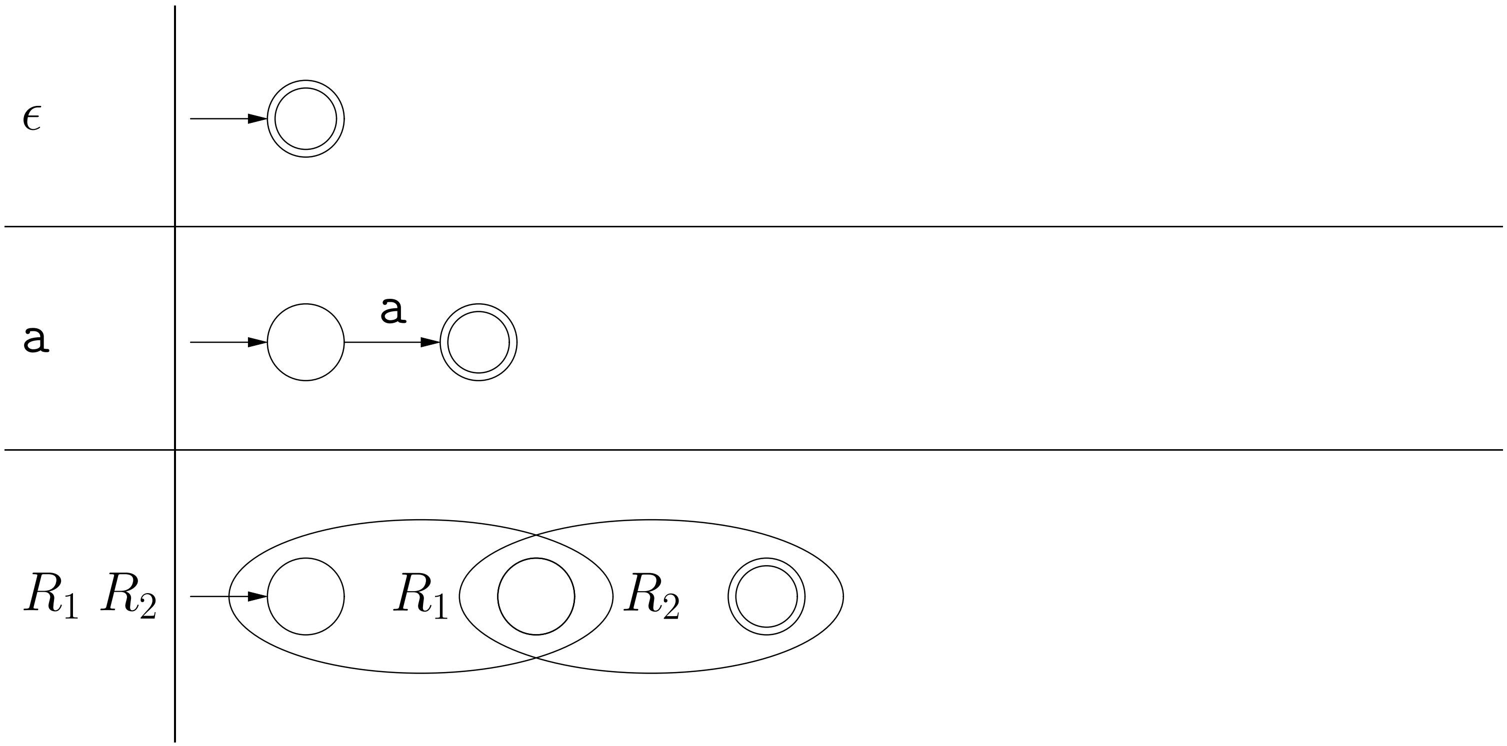
(Edges without labels mean "any character not covered by another edge.")

Example: What does this NFA recognize?



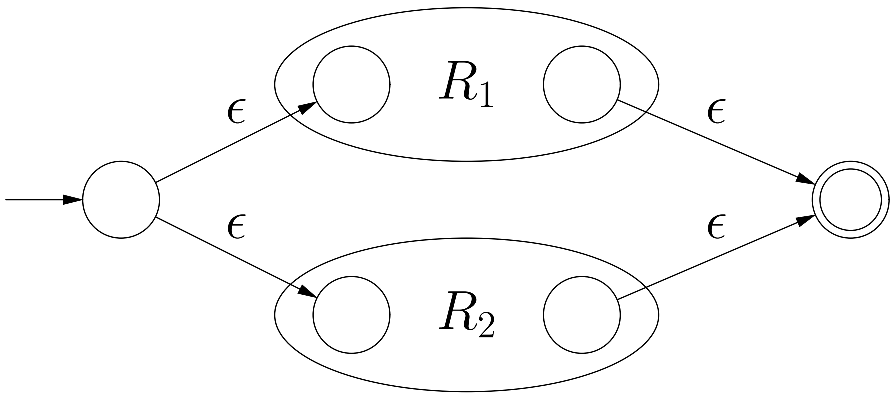
What is the simplest equivalent DFA you can think of?

Review: Classical Regular Expressions to NFAs (I)

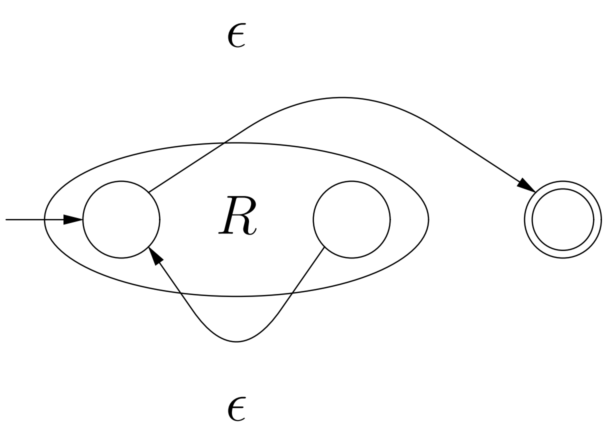


Review: Classical Regular Expressions to NFAs (II)

$R_1 \mid R_2$



R^*



Extensions?

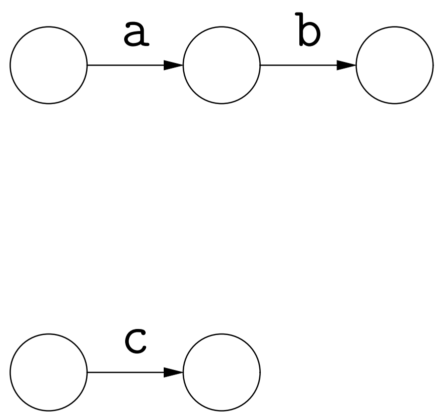
- How would you translate ϕ (the empty language, containing no strings) into an FA?
- How could you translate 'R?' into an NFA?
- How could you translate 'R+' into an NFA?
- How could you translate '$R_1|R_2|\cdots|R_n$' into an NFA?

Example of Conversion

How would you translate `((ab)*|c)*` into an NFA (using the construction above)?

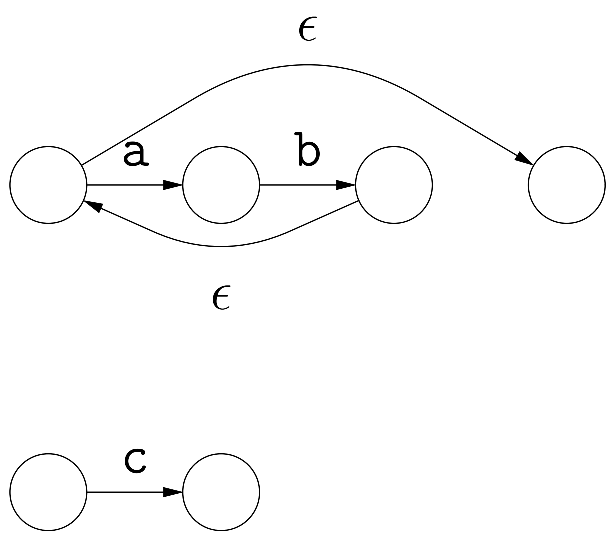
Example of Conversion

How would you translate `((ab)*|c)*` into an NFA (using the construction above)?



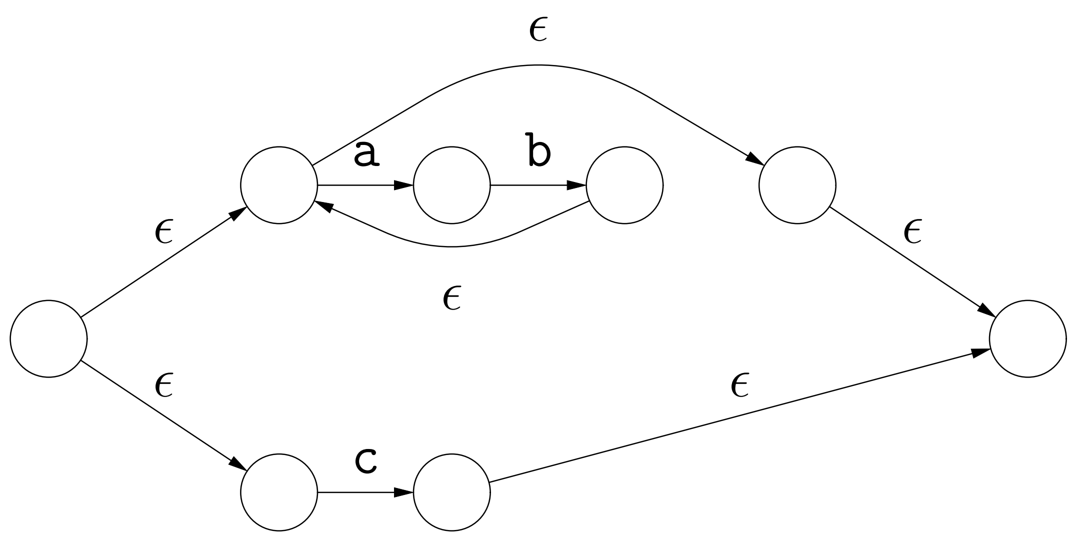
Example of Conversion

How would you translate `((ab)*|c)*` into an NFA (using the construction above)?



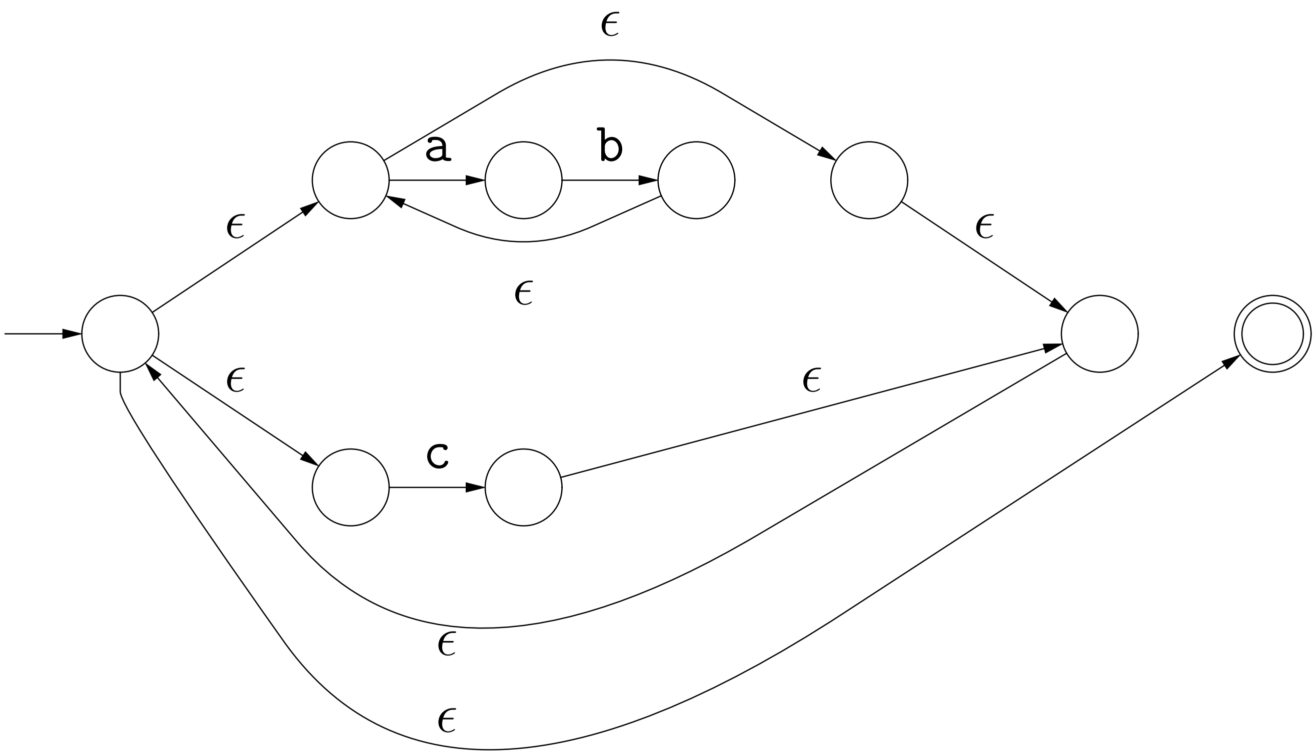
Example of Conversion

How would you translate `((ab)*|c)*` into an NFA (using the construction above)?

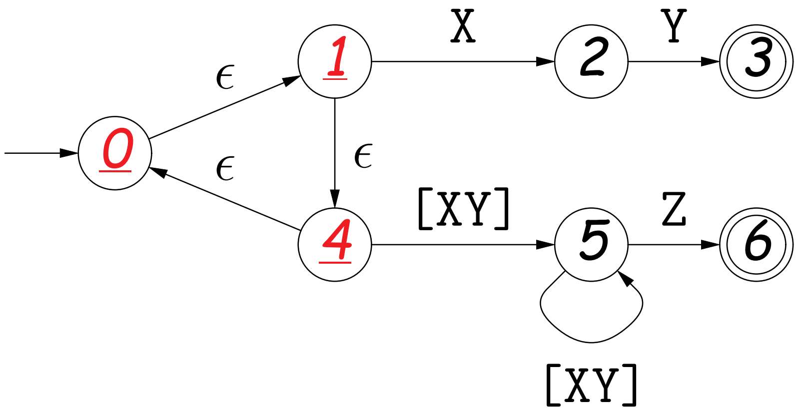


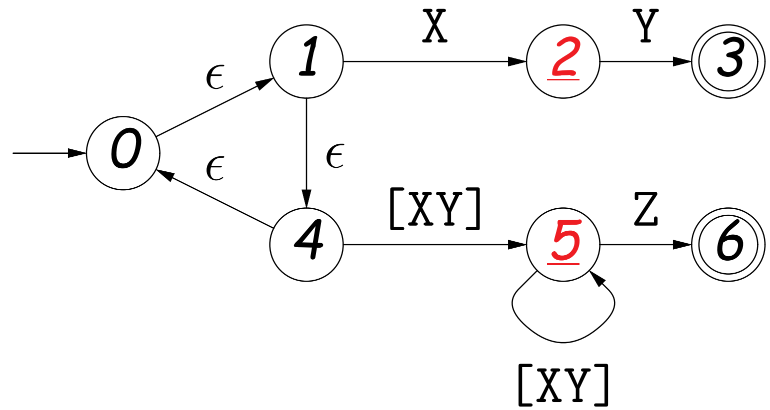
Example of Conversion

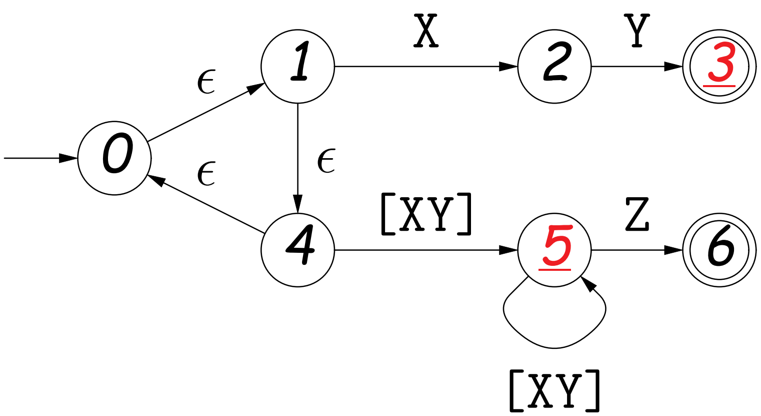
How would you translate `((ab)*|c)*` into an NFA (using the construction above)?

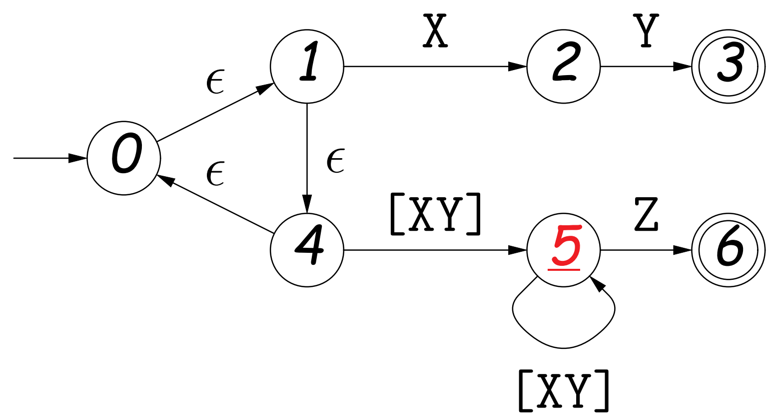


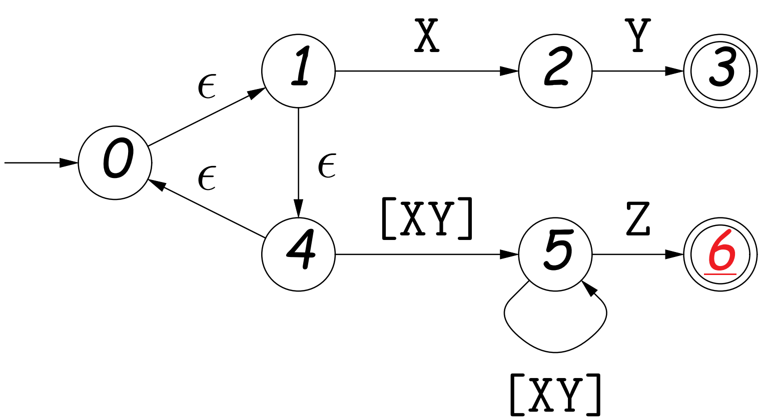
Abstract Implementation of NFAs







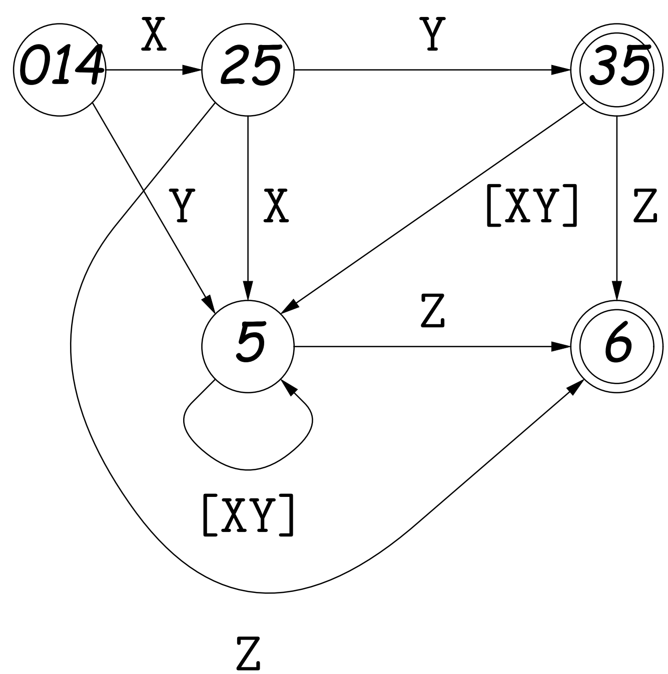




String: XYYZ

Review: Converting to DFAs

- **OBSERVATION:** The set of states that are marked (colored red) changes with each character in a way that depends only on the set and the character.
- In other words, machine on previous slide acted like this DFA:



DFAs as Programs

- Can realize DFA in program with control structure:

```
state = INITIAL;
for (s = input; *s != '\0'; s += 1) {
   switch (state):
   case INITIAL:
      if (*s == 'a') state = A_STATE; break;
   case A_STATE:
      if (*s == 'b') state = B_STATE; else state = INITIAL; break;
   ...
   }
}
return state == FINAL1 || state == FINAL2;
```

- Or with data structure (table driven):

```
state = INITIAL;
for (s = input; *s != '\0'; s += 1)
   state = transition[state][s];
return isfinal[state];
```

What JLex and Flex Do

- A JLex or Flex program specification is a *giant regular expression* of the form $R_1|R_2|\cdots|R_n$, where none of the R_i match ϵ.
- Each final state labeled with some action.
- Converted, by previous methods, into a table-driven DFA.
- But, this particular DFA is used to recognize *prefixes* of the (remaining) input: initial portions that put machine in a final state.
- Which final state(s) we end up in determine action. To deal with multiple actions:
  - Match *longest* prefix ("maximum munch").
  - If there are multiple matches, apply *first* rule in order.

How Do They Do It (I)?

Q: How can we use a DFA to recognize longest match?

How Do They Do It (I)?

Q: How can we use a DFA to recognize longest match?

Answer:

- Use the DFA to scan the input until there is no transition on the current symbol.
- Every time the DFA enters a final state, record the input position and the state.
- When the scan stops, reset the input position to the last one saved, and use the last-saved final state as the result.

How Do They Do It (II)?

Q: How can we use DFA to act on the first of equal-length matches? Example:

```
while|[a-zA-Z]+
```

That is, we want our DFA to distinguish the keyword "while" from non-keyword identifiers.

How Do They Do It (II)?

Q: How can we use DFA to act on the first of equal-length matches?
Example:

```
while|[a-zA-Z]+
```

That is, we want our DFA to distinguish the keyword "while" from non-keyword identifiers.

Answer:

- 1. The NFA for patterns of the form $R_1|R_2|\cdots|R_n$ may be formed from the NFAs for each of the R_is.
- 2. In those NFAs, label the final state for R_i the integer i.
- 3. Take the labels of the DFA to be sets of states from the NFA.
- 4. When we determine the final state of the DFA, look at its label and find the smallest of the integer labels from step 2 among the NFA states that label it.

How Do They Do It (III)?

Q: How can we use a DFA to handle the R_1/R_2 pattern (matches just R_1 but only if followed by R_2, like R_1(?=R_2) in Python)?

How Do They Do It (III)?

Q: How can we use a DFA to handle the R_1/R_2 pattern (matches just R_1 but only if followed by R_2, like R_1`(?=`R_2`)` in Python)?

Answer:

- Construct the NFAs for R_1 and R_2 and glue them together to get an NFA for R_1R_2.
- When scanning the string, record the state and position whenever you pass through a final state of the original R_1.
- When you get to a final state of the combined pattern for R_1R_2, use the last recorded final state and position for R_1.