Implementation of Lexical Analysis

Lecture 4

Tips on Building Large Systems

- KISS (Keep It Simple, Stupid!)
- Don't optimize prematurely
- Design systems that can be tested
- It is easier to modify a working system than to get a system working

Outline

- Specifying lexical structure using regular expressions
- Finite automata
  - Deterministic Finite Automata (DFAs)
  - Non-deterministic Finite Automata (NFAs)
- Implementation of regular expressions

  RegExp => NFA => DFA => Tables

Notation

- There is variation in regular expression notation

- Union: $A \mid B \equiv A + B$
- Option: $A + \varepsilon \equiv A?$
- Range: 'a' +' b' +...+' z' $\equiv$ [a-z]
- At least one: A^+ (A-Abdullah) $\equiv AA^*$
- Excluded range:

  complement of [a-z] $\equiv$ [^a-z]

Regular Expressions in Lexical Specification

- Last lecture: a specification for the predicate

$$s \in L(R)$$

Note: we can do this just by looking at reg. exp. R

- But a yes/no answer is not enough!
  - Because we need to know not just whether the string is a valid program, but also…
- How to partition the input into tokens

- We adapt regular expressions to this goal
  - I.e., there are some small required extensions

Regular Expressions => Lexical Spec. (1)

1. Write a regexp for the lexemes of each token class
   - Number = digit +
   - Keyword = 'if' + 'else' + ...
   - Identifier = letter (letter + digit)*
   - OpenPar = '('
   - ...

- So, we write down a whole list of regular expressions, one for each syntactic category in language

Regular Expressions => Lexical Spec. (2)

2. Construct R, matching all lexemes for all token classes

   $R = \text{Keyword} + \text{Identifier} + \text{Number} + \ldots$

   $= R_1 + R_2 + \ldots$

- What follows is the key to how we use the regular expression specification to perform lexical analysis

Regular Expressions => Lexical Spec. (3)

3. Let input be $x_1 \ldots x_n$

   For $1 \leq i \leq n$ check whether the prefix

   $$x_1 \ldots x_i \in L(R)$$

4. If success, then we know that

   $x_1 \ldots x_i \in L(R_j)$ for some j

5. Remove $x_1 \ldots x_i$ from input and go to (3)

Continue removing pieces until we have tokenized the entire string

Ambiguities (1)

- There are ambiguities in the algorithm
  - Some things are under specified (and these turn out to be interesting)
- How much input is used? What if
    - $x_1 \ldots x_i \in L(R)$ and also
    - $x_1 \ldots x_K \in L(R)$ (of course $i \neq k$)
    - Ex. We've got ==
- Rule: Pick longest possible string in $L(R)$
  - The "maximal munch"
    - Yes, this is really what this rule is called

Do we see "How" or H-o-w?

Ambiguities (1)

- There are ambiguities in the algorithm
  - Some things are under specified (and these turn out to be interesting)
- How much input is used? What if
  - $x_1 \ldots x_i \in L(R)$ and also
  - $x_1 \ldots x_K \in L(R)$ (of course $i \neq k$)
  - Ex. We've got ==
- Rule: Pick longest possible string in $L(R)$
  - The "maximal munch"
    - Reason is that this is the way humans read things
    - So tools work this way as well (which usually does right thing)

Ambiguities (2)

- Which token is used? What if
  - $x_1 \ldots x_i \in L(R_j)$ and also
  - $x_1 \ldots x_i \in L(R_k)$

- Ex. Recall our specifications for keywords and identifiers
  - Keyword = 'if' + 'else' + …
  - Identifier = letter (letter + digit)*
  - 'if' satisfies both

Ambiguities (2)

- Which token is used? What if
  - $x_1 \ldots x_i \in L(R_j)$ and also
  - $x_1 \ldots x_i \in L(R_k)$

- Note: in most languages, if it's a keyword, it's not an identifier
  - But: changing RE for Identifier to explicitly exclude keywords is a real pain (current rule much more natural)

Ambiguities (2)

- Which token is used? What if
  - $x_1 \ldots x_i \in L(R_j)$ and also
  - $x_1 \ldots x_i \in L(R_k)$
- Rule: use rule listed first (j if $j < k$)
  - Treats "if" as a keyword, not an identifier
  - Bottom line: in file defining our lexical specification, put Keywords before the Identifiers

Error Handling

- What if

  No rule matches a prefix of input ?

  Note: This comes up quite a bit
- Problem: Can't just get stuck …
  - I.e., Important for compiler to do good error handling (can't simply crash)
  - Need to provide feedback as to where the error is and what kind of error it is

Error Handling

- What if

  No rule matches a prefix of input ?

  Note: This comes up quite a bit

- Solution:
  - Don’t let it ever happen that a string isn’t in L(R)
  - ???!!!

Error Handling

- What if

  No rule matches a prefix of input ?

  Note: This comes up quite a bit
- Solution:
  - Don't let it ever happen that a string isn't in L(R)
  - Write a rule matching all "bad" strings
    - Create an Error token class
  - Put it last (lowest priority)
    - Putting it last also allows us to be a little bit sloppy – can include strings in the RE that ARE valid
    - Earlier rules will have caught these
    - Action for this rule is to print an error string

Summary

- Regular expressions provide a concise notation for string patterns

- Use in lexical analysis requires small extensions
  - To resolve ambiguities
  - To handle errors

- Warning: When you actually go to write the specification for a lexor, the two rules for resolving ambiguity can lead to tricky situations – you must think carefully about the ordering of the rules!
  - You may not always be getting what you think you are!

Summary

- Regular expressions provide a concise notation for string patterns

- Use in lexical analysis requires small extensions
  - To resolve ambiguities
  - To handle errors

- Good algorithms known
  - Require only single pass over the input
  - Few operations per character (table lookup)
  - These algorithms are the subject of the following slides

Finite Automata

- Regular expressions = specification
- Finite automata = implementation

Closely related: they can specify exactly the same languages -- the *regular languages*

Finite Automata

- Regular expressions = specification
- Finite automata = implementation

Closely related: they can specify exactly the same languages -- the *regular languages*

We won't prove this, but we will use it

Finite Automata

- Regular expressions = specification
- Finite automata = implementation

- A finite automaton consists of
  - An input alphabet Σ (characters the FA can read)
  - A finite set of states S (thus "finite" automata)
  - A start state n
  - A set of accepting states $F \subseteq S$
  - A set of transitions $\text{state} \rightarrow^{\text{input}} \text{state}$
    - I.e., if it's in a given state, it can read some input and move to another specified state

Finite Automata

- Transition

$$s_1 \rightarrow^{a} s_2$$

- Is read

  In state s_1 on input "a" go to state s_2

- If end of input and in accepting state => accept
  - That is, "yes, this string was in the language of this machine"

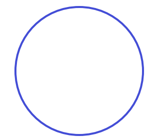
- Otherwise => reject

So

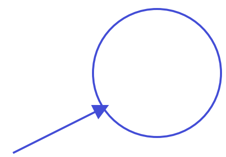
- Start in start state
- Repeat (until end of input string):
  - Read one character of input string
  - Move to appropriate state
- If after last character read you end up in accepting state, string is in language of the FA
- Else, string not in language of FA
  - E.g., Terminates in state not in F or
  - Machine gets stuck: finds itself in a state and there is no transition of that state on the input (note that it does not read "out of")

Alternative Notation: Finite Automata State Graphs

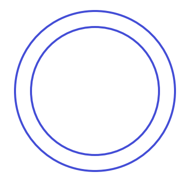
- A state



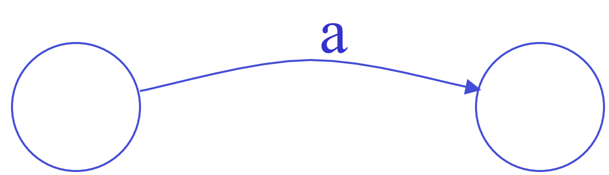
- The start state



- An accepting state

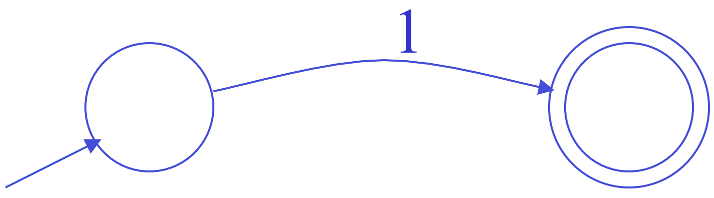


- A transition



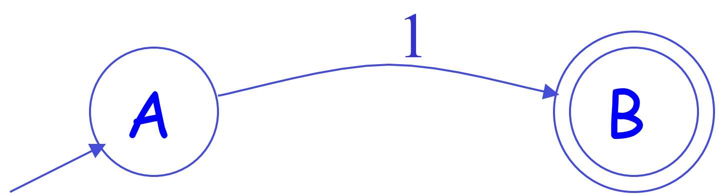
A Simple Example

- A finite automaton that accepts only “1”



How The Machine Executes

- A finite automaton that accepts only “1”

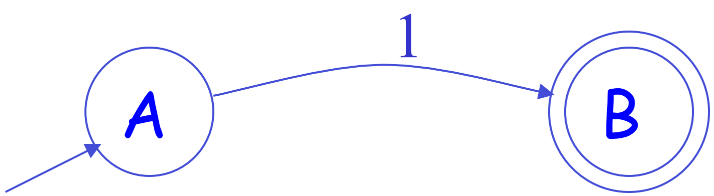


StateState	Input
A	1
B	1

Input pointer always advances one spot. Never moves backwards

How The Machine Executes

- A finite automaton that accepts only "1"



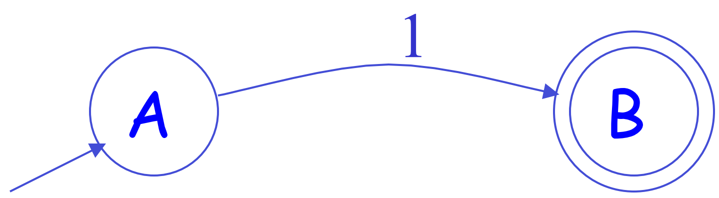
StateState **Input**

A 0

0

How The Machine Executes

- A finite automaton that accepts only “1”



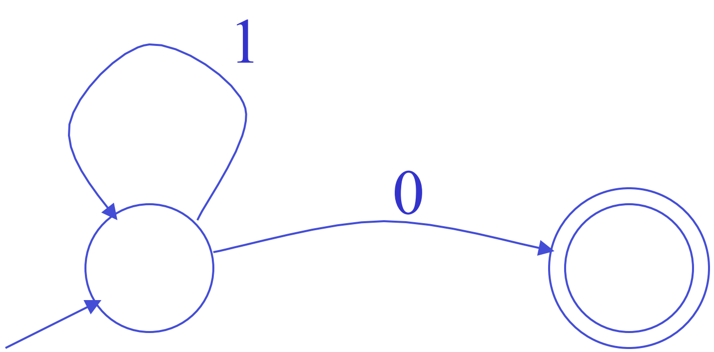
StateState	Input
A	1 0
B	1 0
	1 0

The Language of a Finite Automata

- Is the set consisting of accepted strings

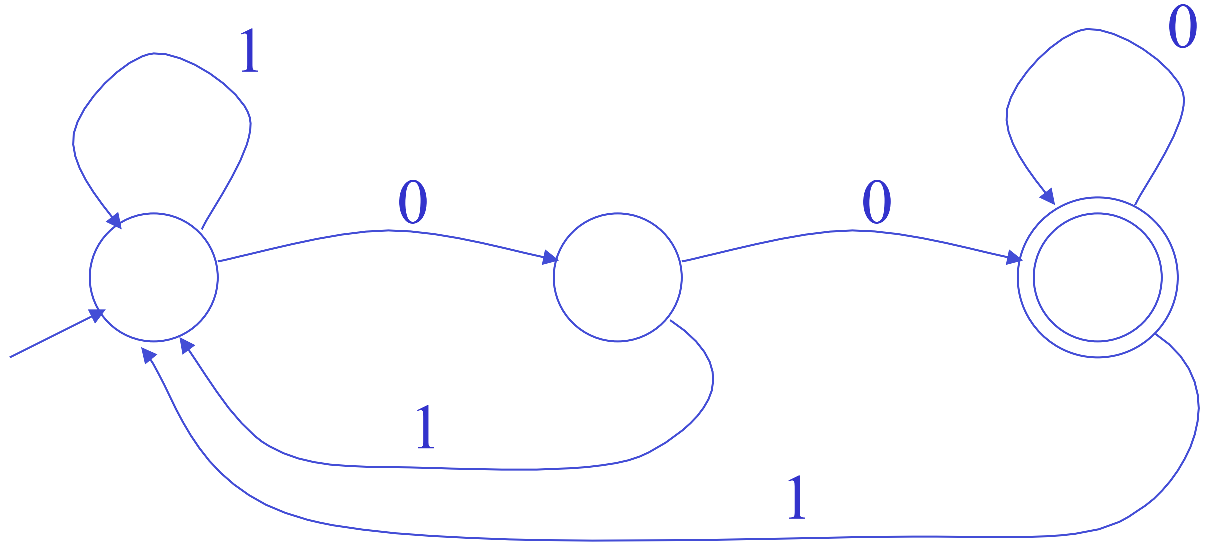
Another Simple Example

- A finite automaton accepting any number of 1's followed by a single 0
- Alphabet: {0,1}



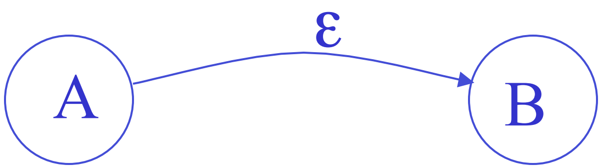
And Another Example

- Alphabet {0,1}
- What language does this recognize?



Epsilon Moves

- Another kind of transition: ε-moves



- Machine can move from state *A* to state *B* without reading input

Deterministic and Nondeterministic Automata

- Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No ε-moves
- Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have ε-moves

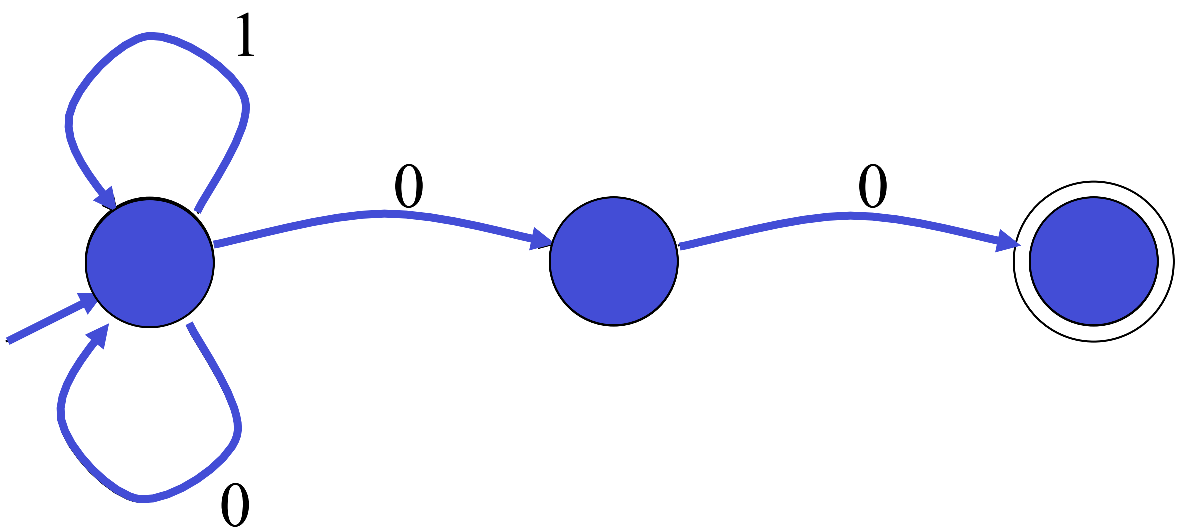
Execution of Finite Automata

- A DFA can take only one path through the state graph
  - Completely determined by input

- NFAs can choose
  - Whether to make ε-moves
  - Which of multiple transitions for a single input to take

Acceptance of NFAs

- An NFA can get into multiple states



- Input: 1 0 0

Rule: NFA accepts if it <u>can</u> get to a final state

NFA vs. DFA (1)

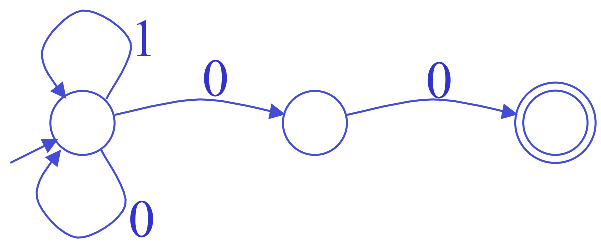
- NFAs and DFAs recognize the same set of languages (regular languages)
  - Think of NFAs as parallel processors

- DFAs are faster to execute
  - There are no choices to consider

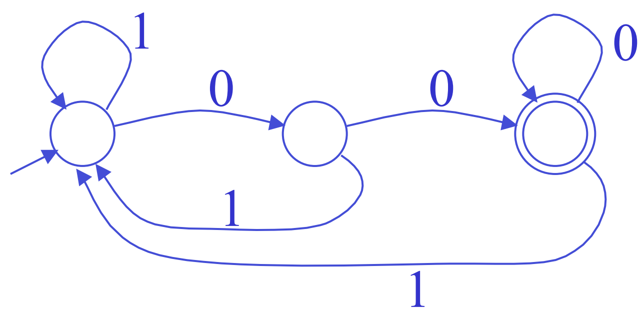
NFA vs. DFA (2)

- For a given language NFA can be simpler than DFA

NFA



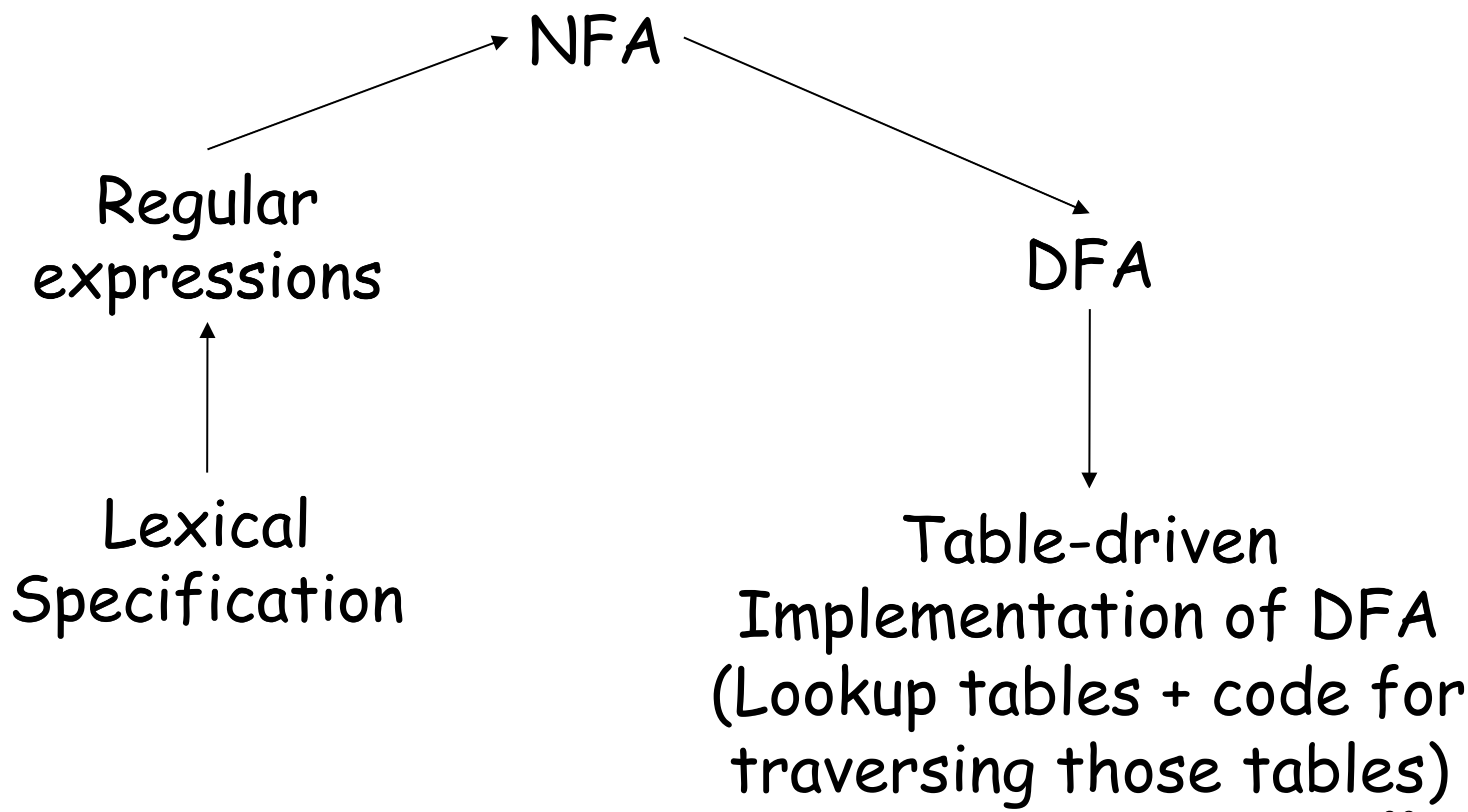
DFA



- DFA can be exponentially larger than NFA (why would you expect that to be the case?)

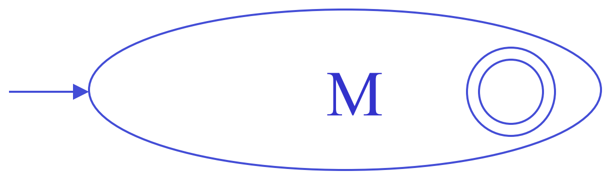
Regular Expressions to Finite Automata

- High-level sketch

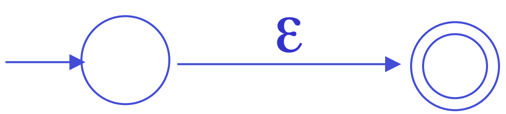


Regular Expressions to NFA (1)

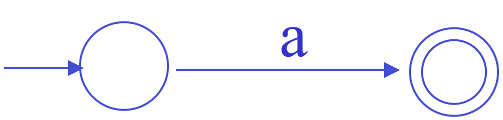
- For each kind of rexp, define an equivalent NFA
  - Notation: NFA for rexp M



- For ε

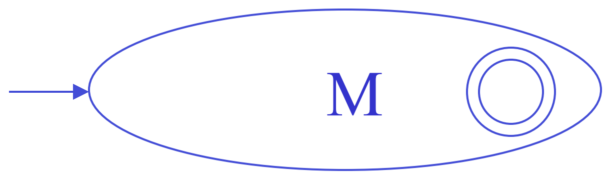


- For input a

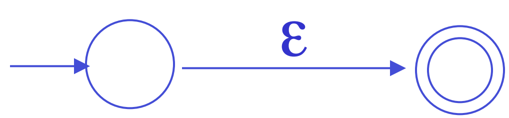


Regular Expressions to NFA: Atomic REs

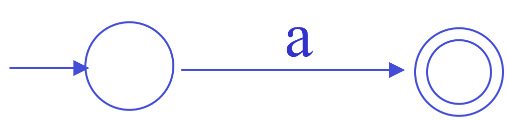
- For each kind of rexp, define an equivalent NFA
  - Notation: NFA for rexp M



- For ε



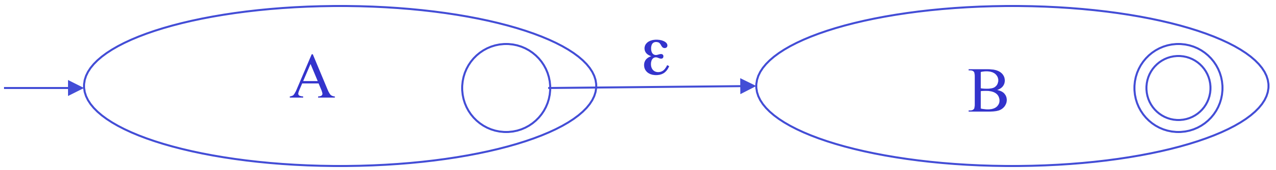
- For input a



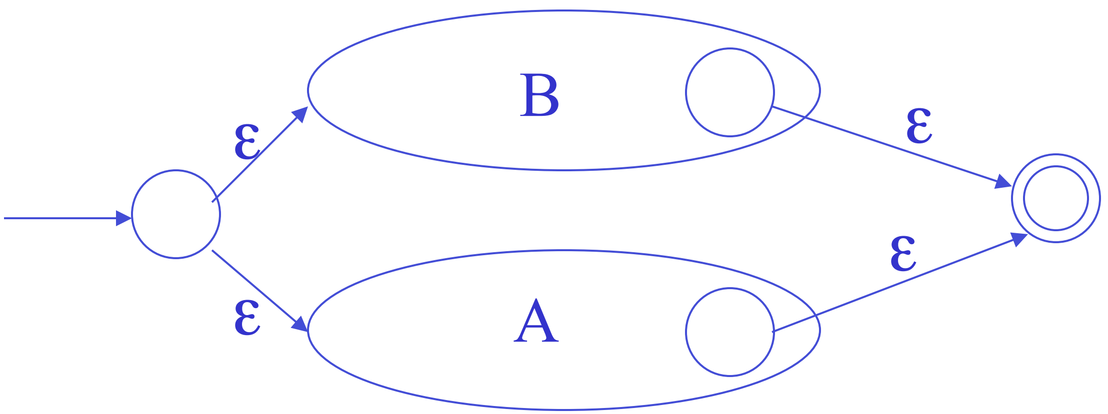
Notion here is that we will be building our overall machine up from smaller machines

Regular Expressions to NFA: Compound REs

- For *AB*



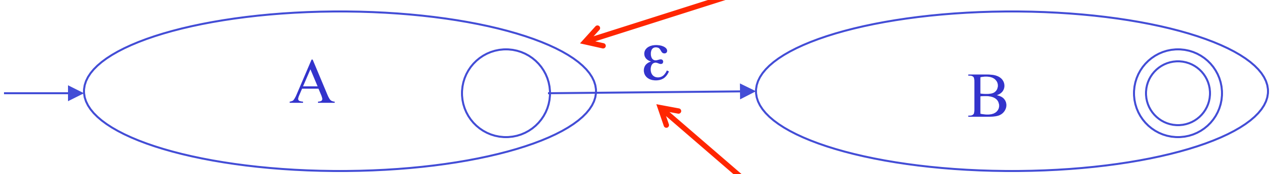
- For *A* + *B*



Regular Expressions to NFA: Compound REs

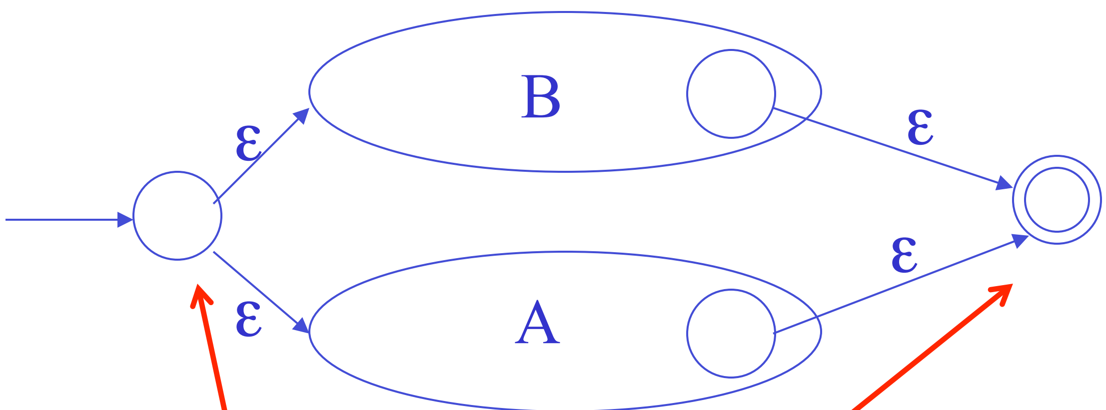
- For *AB*

Note modification of final state



And addition of ε transition

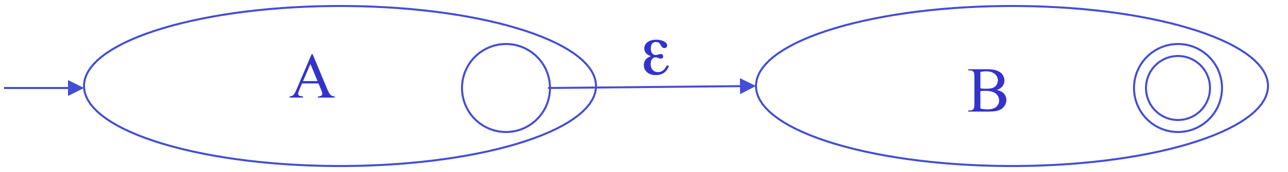
- For *A + B*



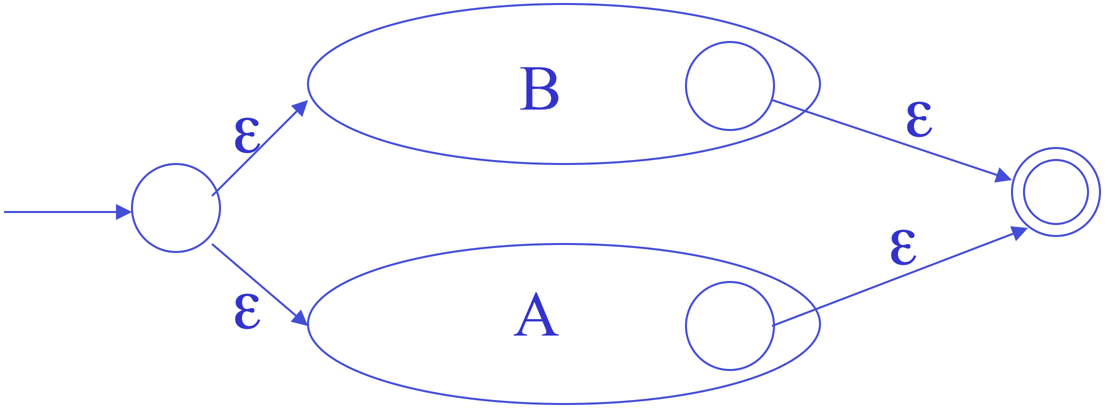
Note addition of new start and final states

Regular Expressions to NFA: Compound REs

- For *AB*



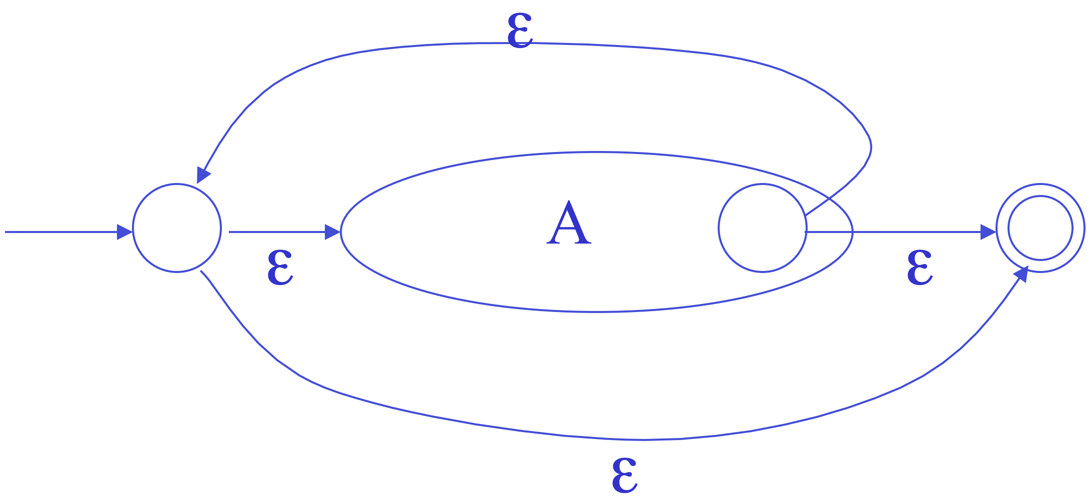
- For *A + B*



Remember: With NFA if **ANY** choice works, string is accepted

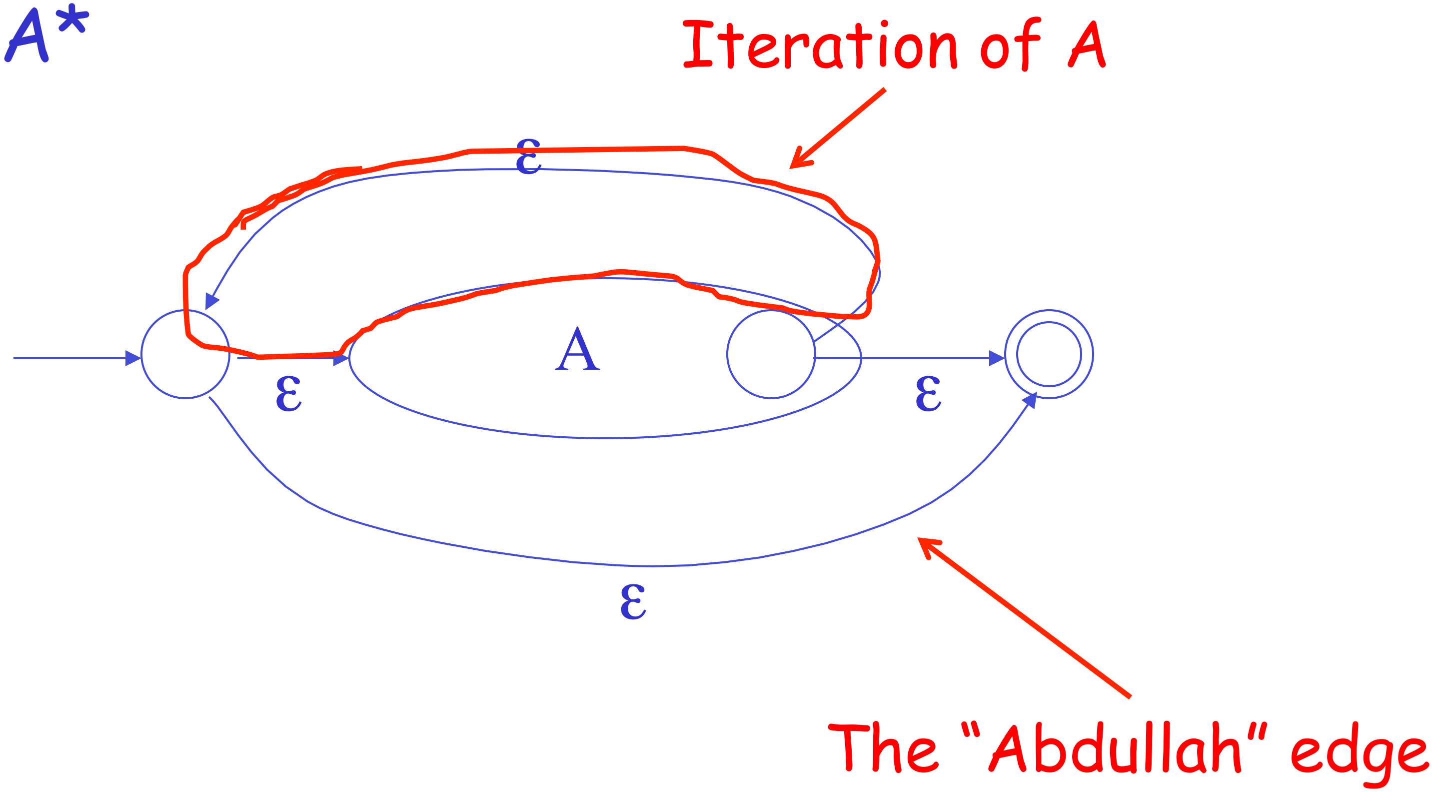
Regular Expressions to NFA (3)

- For *A**



Regular Expressions to NFA (3)

- For *A**



Let’s remember one very important thing in all of this: it is done this way so that the process can be automated!

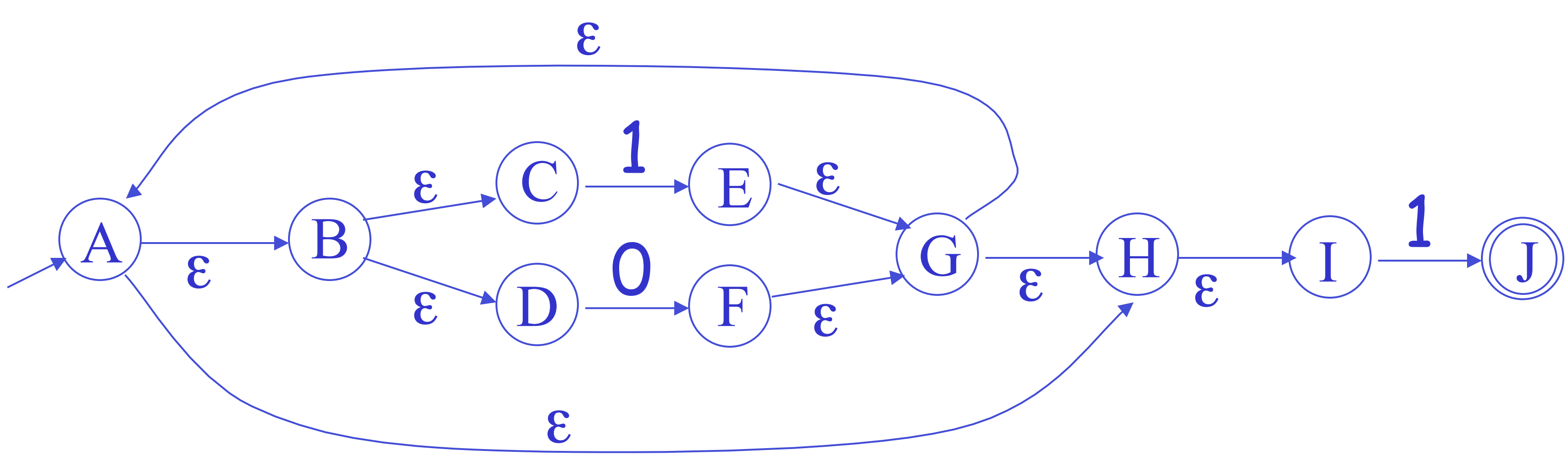
You might see easier DFAs or NFAs, but a computer needs to have an algorithm to create these, which is why we go through this process.

Example of RegExp -> NFA conversion

- Consider the regular expression

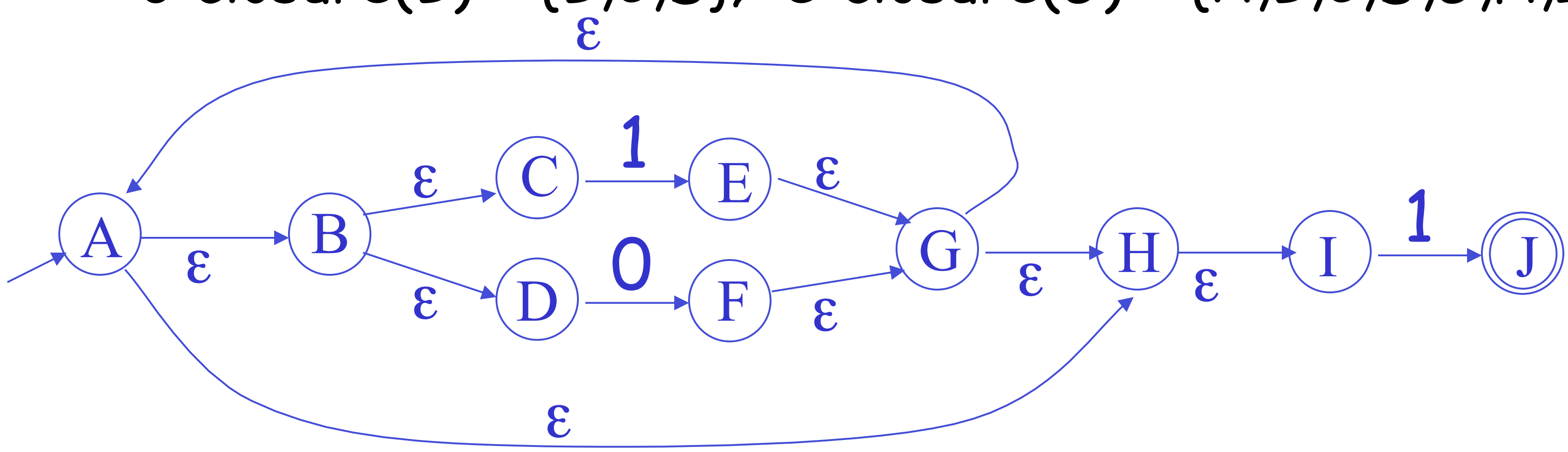
  (1+0)*1

- The NFA is



ε-closure

- An idea that helps with transition from NFA to DFA
- ε-closure of state B is all states that can be reached from B via epsilon moves
  - ε-closure(B) = {B,C,D}; e-closure(G) = {A,B,C,D,G,H,I}



NFA to DFA. Remark

- An NFA may be in many states at any time
- How many different states ?
- If there are N states, the NFA must be in some subset of those N states
- How many subsets are there?
  - $2^N - 1$ = finitely many

Some helpful notation

- For a character a in the input language Σ, and $\mathcal{X}$ a set of states in the NFA, define a($\mathcal{X}$) to be a subset of the set of all states in the NFA defined as follows:

- a($\mathcal{X}$) = {Y | for some X in $\mathcal{X}$, there is a transition from X to Y on input a}
  - I.e., If we are in some state in set $\mathcal{X}$, and a is the next input, then a($\mathcal{X}$) is the set of states to which we could transition.

NFA to DFA: *The Trick*

- Simulate the NFA
- Each state of DFA is a **non-empty** subset of states of the NFA
  - Though not all subsets of states of the NFA will have links to/from them
  - So a large number of possible states (but finite!)

NFA to DFA: *The Trick*

- Start state
  - the set of NFA states reachable through ε-moves from NFA start state
  - ε-clos(start state)

- Think about why this makes sense: which sets of states might the NFA be in at the beginning of execution?

NFA to DFA: *The Trick*

- Add a transition $S \rightarrow^{a} S'$ to DFA iff
  - S' is the set of NFA states reachable from any state in S after seeing the input a, considering ε-moves as well
  - ε-clos(a(S))

NFA to DFA: *The Trick*

- Final states
  - the set of DFA states X such that X contains a state in F (the set of final states of the NFA)

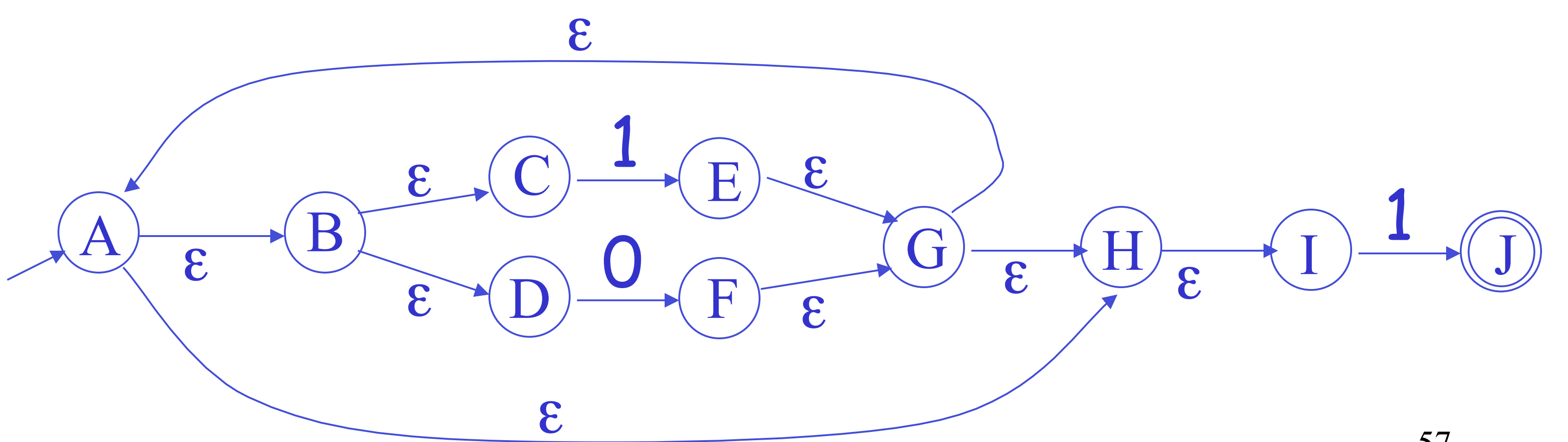
- Think about why this makes sense: if a subset of NFA states contains a state that was in F (for the NFA), and execution of the DFA ends up in that state, then there is a path through the NFA which ends in a final state.

Is What We Get a DFA?

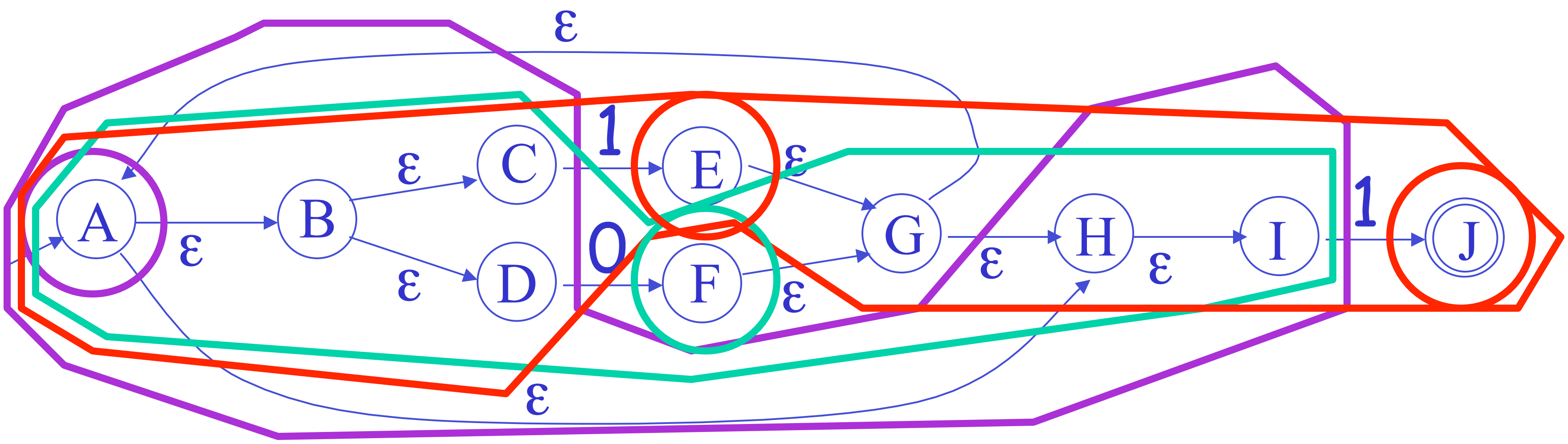
- A finite set of states
- A start state
- A set of final states
- A transition function with only one move per input
- No ε-moves

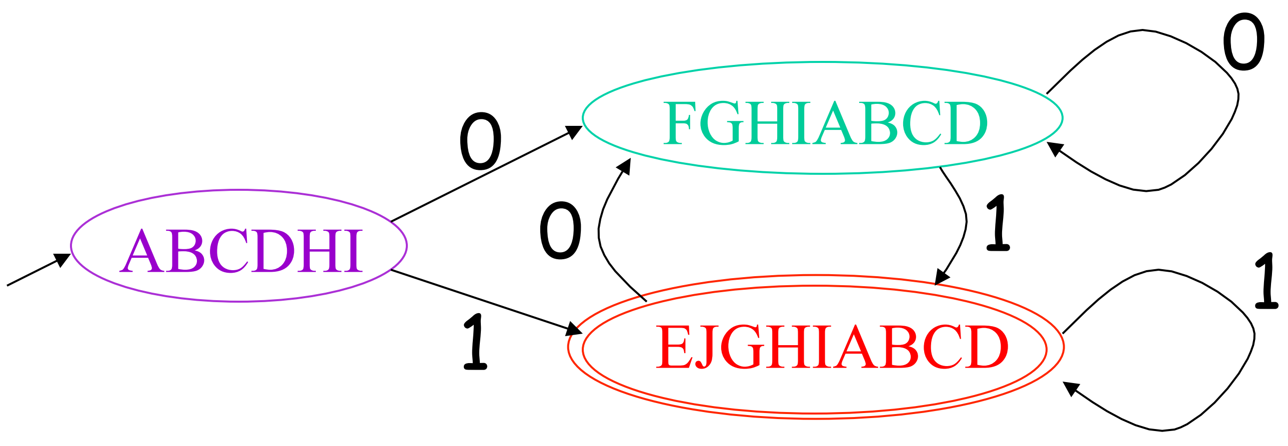
Let's do this!

- We won't follow exact steps: too many possible states to list them all (and most not involved in final DFA)
- Start with start state, then work out which additional states required



NFA -> DFA Example





The Final Step: Implementation of DFA

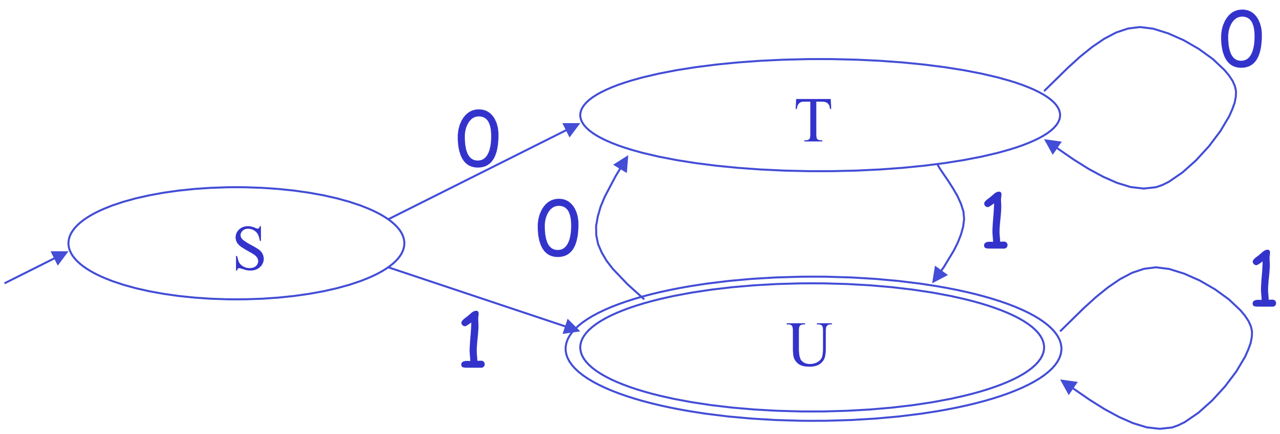
- First, note that our original diagram was somewhat misleading. Some systems go straight from NFA to implementation
  - More on this in a bit

Implementation

- A DFA can be implemented by a 2D table T
  - One dimension is "states"
  - Other dimension is "input symbol"
  - For every transition $S_i \rightarrow^a S_k$ define $T[i,a] = k$

- DFA "execution"
  - If in state S_i and input a, read $T[i,a] = k$ and skip to state S_k
  - Very efficient

Table Implementation of a DFA



input symbol

state

	0	1
S	T	U
T	T	U
U	T	U

The code

```
char[] input;
int i = 0;
int state = 0;

while ( input[i] != '\0' ) {
    state = transitionTable[ state, input[i]];
    ++i;
}
```

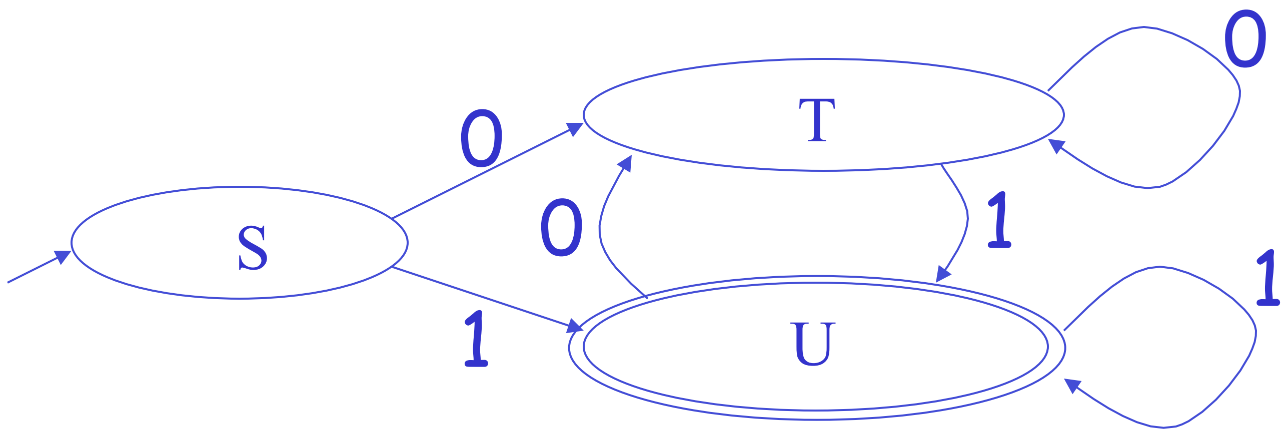
The code

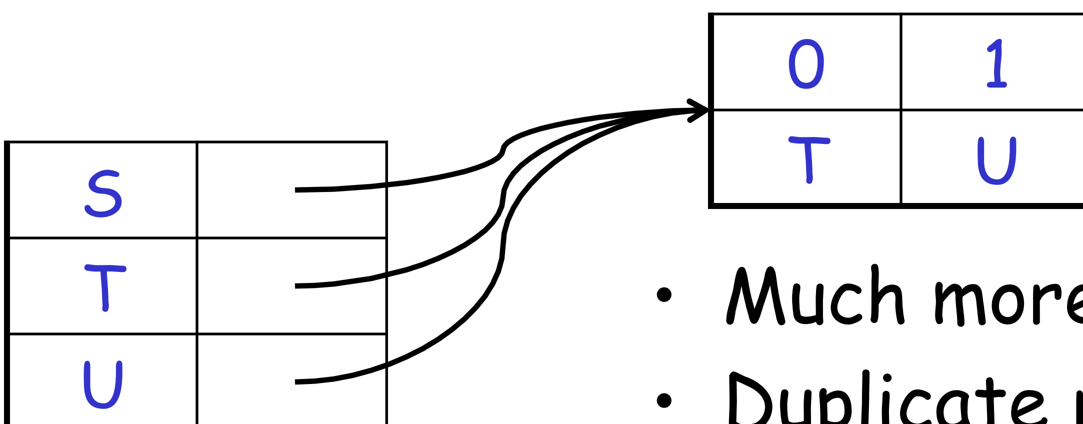
```
char[] input;
int i = 0;
int state = 0;

while ( input[i] != '\0' ) {
    state = transitionTable[ state, input[i]];
    ++i;
}
```

Note how compact and efficient this is

Table Implementation of a DFA





- Much more compact
- Duplicate rows common in DFAs for LA

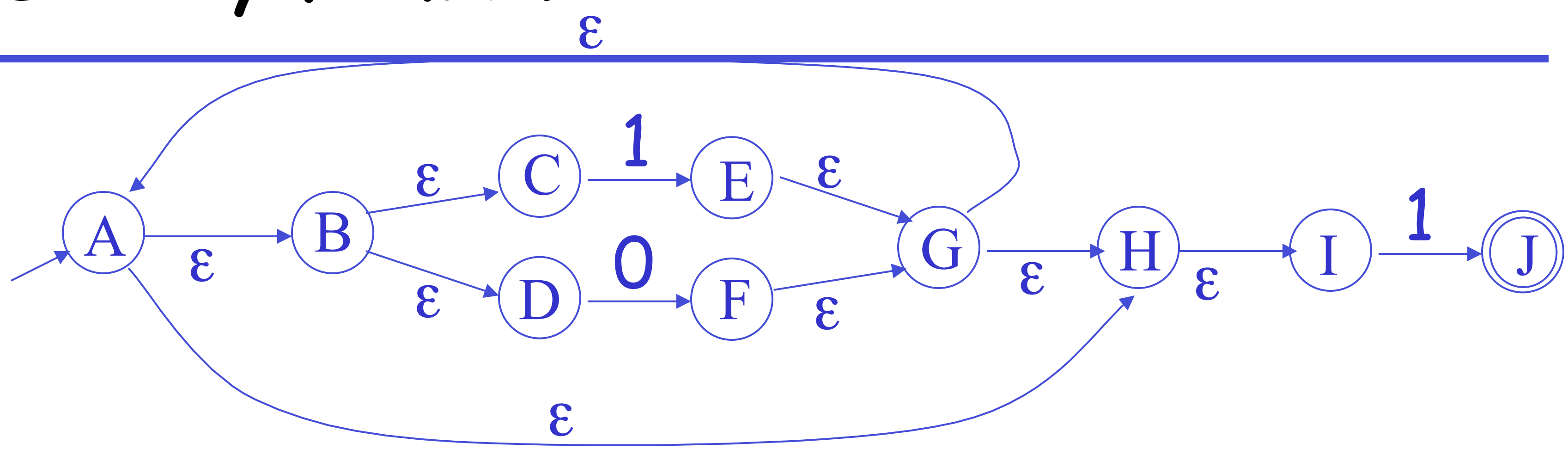
Compaction of Table

- Turns out that this technique can make tables much more compact
- Moreover, duplicate rows show up quite a bit in the FAs that arise in lexical analysis
- Considering large number of potential states, this can lead to considerable compaction of the table
  - While blowup in states from NFA to DFA is often not the worst case, it can be substantial
  - 2D table can be quite large
- Disadvantage: pointer dereferences

What if we don't want a DFA?

- Why? Table that results might be huge, so may want to use NFA directly

Directly from NFA to Table



	0	1	ε
A			{B,H}
B			{C,D}
C		E	
D	F		

Good and Bad of Going Directly from NFA to Table

- Good: Table guaranteed to be relatively small
  - limited by size of NFA and size of input alphabet
- Bad: Even with compression tricks, inner loop runs much more slowly because dealing with sets of states, rather than states themselves
  - So on each move, we need to keep track of all possible states to which we could go (and associated ε-moves)
- Bottom line: Can save a lot of space, but cost a lot in time

Implementation (Cont.)

- NFA -> DFA conversion is at the heart of tools such as flex
- But, DFAs can be huge
- In practice, flex-like tools trade off speed for space in the choice of NFA and DFA representations
  - User chooses via configuration, whether they want to be closer to full DFA or full NFA