Lecture 3
Implementation of Lexical Analysis

Review: Last Written Assignment

- Write regular expressions for the following languages over the alphabet $\Sigma = \{0, 1\}$:
  - (a) The set of all strings which start and end with the same digit.
  - (b) The set of all strings representing a binary number where the sum of its digits is even.
  - (c) The set of all strings that contain the substring 10100.

Answers

- (a) The set of all strings which start and end with the same digit.

  $1(0^* 1)^* + 0(1^* 0)^*$

- (b) The set of all strings representing a binary number where the sum of its digits is even.

  $(0^* 10^* 1)^* 0^*$

- (c) The set of all strings that contain the substring 10100.

  $(0+1)^* 10100(0+1)^*$

Yours - (a)

- $0(0+1)^*0 + 1(0+1)^*1$

Yours - (b)

- 0*(10*1）*0*
- (10)*1+0)*

- ((10*1)+0)*
- (10*1)*0* (10*1)*0* (10*1)*
- 0*+(0*+(10*1)*)*
- (0*10*10*)*+0*
- ((10*1)+0*)*
- ((0*(11)*0*)* + (0*10*10*)*)*

Next: 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]
- Excluded range: complement of [a-z] $\equiv$ [^a-z]

Regular Expressions in Lexical Specification

- Last lecture: a specification for the predicate $s \in L(R)$

- But a yes/no answer is not enough!
- Instead: partition the input into tokens
- We adapt regular expressions to this goal

Regular Expressions => Lexical Spec. (1)

1. Select a set of tokens
   - Number, Keyword, Identifier, ...

2. Write a R.E. for the lexemes of each token
   - Number = $digit^+$
   - Keyword = 'if' | 'else' | ...
   - Identifier = letter (letter | digit)*
   - OpenPar = '('
   - ...

Regular Expressions => Lexical Spec. (2)

3. Construct R, matching all lexemes for all tokens

$$R = \text{Keyword} \mid \text{Identifier} \mid \text{Number} \mid \ldots$$
$$= R_1 \mid R_2 \mid R_3 \mid \ldots$$

Facts: If $s \in L(R)$ then s is a lexeme

- Furthermore $s \in L(R_i)$ for some "i"
- This "i" determines the token that is reported

Regular Expressions => Lexical Spec. (3)

4. Let the input be $x_1 \ldots x_n$

   ($x_1 \ldots x_n$ are characters in the language alphabet)

   - For $1 \leq i \leq n$ check

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

5. If success, then we know that

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

6. Remove $x_1 \ldots x_i$ from input and go to (4)

Lexing Example

R = Whitespace | Integer | Identifier | '+'

- Parse "f+3 +g"
  - "f" matches R, more precisely Identifier
  - "+" matches R, more precisely '+'
  - ...
  - The token-lexeme pairs are
    (Identifier, "f"), ('+', "+"), (Integer, "3")
    (Whitespace, " "), ('+', "+"), (Identifier, "g")
- We would like to drop the Whitespace tokens
  - after matching Whitespace, continue matching

Ambiguities (1)

- There are ambiguities in the algorithm
- Example:

  R = Whitespace | Integer | Identifier | '+'
- Parse "foo+3"
  - "f" matches R, more precisely Identifier
  - But also "fo" matches R, and "foo", but not "foo+"
- How much input is used? What if
    - $x_1 \ldots x_i \in L(R)$ and also $x_1 \ldots x_K \in L(R)$
  - "Maximal munch" rule: *Pick the longest possible substring that matches R*

More Ambiguities

R = Whitespace | 'new' | Integer | Identifier

- Parse "new foo"
  - "new" matches R, more precisely 'new'
  - but also Identifier, which one do we pick?
- In general, if $x_1 \ldots x_i \in L(R_j)$ and $x_1 \ldots x_i \in L(R_k)$
  - Rule: use rule listed first (j if j < k)

- We must list 'new' before Identifier

Error Handling

R = Whitespace | Integer | Identifier | '+'

- Parse "=56"
  - No prefix matches R: not "=", nor "=5", nor "=56"
- Problem: What if no rule matches a prefix of input? Can't just get stuck …
- Solution:
  - Add a rule matching all "bad" strings; and put it last
- Lexer tools allow the writing of:

  $R = R_1 \mid \ldots \mid R_n \mid \text{Error}$

  - Token Error matches if nothing else matches

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)

Finite Automata

- Regular expressions = specification
- Finite automata = implementation

- A finite automaton consists of
  - An input alphabet Σ
  - A set of states S
  - A start state n
  - A set of accepting states $F \subseteq S$
  - A set of transitions $\text{state} \to^{\text{input}} \text{state}$

Finite Automata

- Transition

$$s_1 \to^a s_2$$

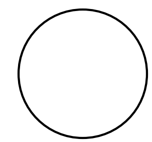
- Is read

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

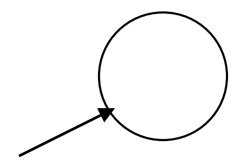
- If end of input and in accepting state => accept
- Othewise => reject

Finite Automata State Graphs

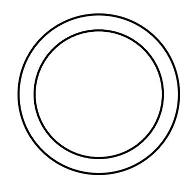
- A state



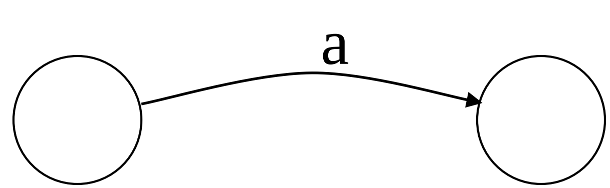
- The start state



- An accepting state

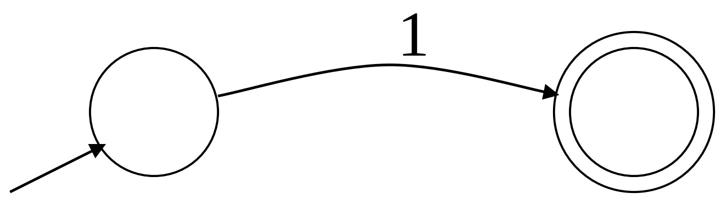


- A transition



A Simple Example

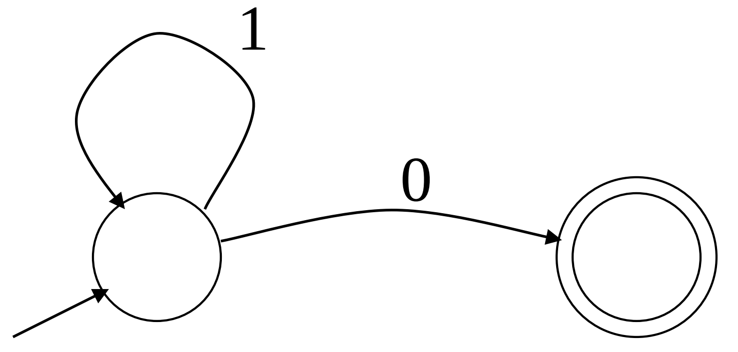
- A finite automaton that accepts only "1"



- A finite automaton accepts a string if we can follow transitions labeled with the characters in the string from the start to some accepting state

Another Simple Example

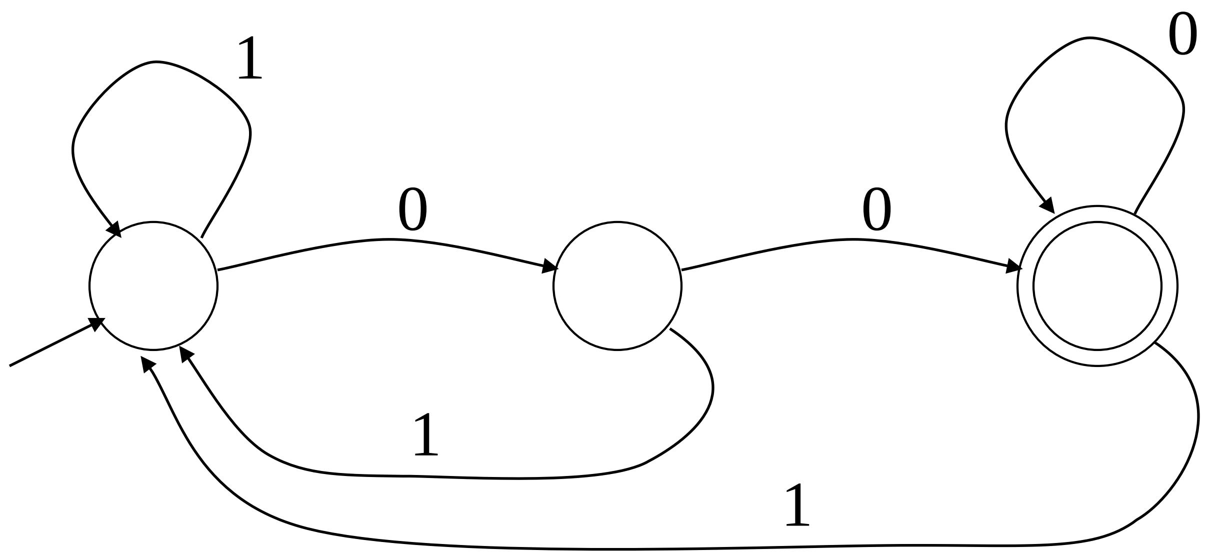
- A finite automaton accepting any number of 1's followed by a single 0
- Alphabet: {0,1}



- Check that "1110" is accepted but "110…" is not

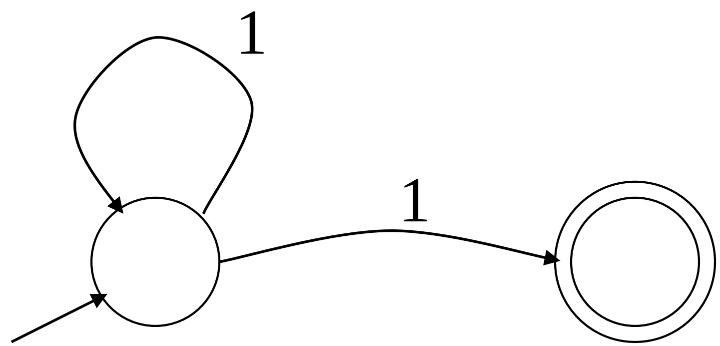
And Another Example

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



And Another Example

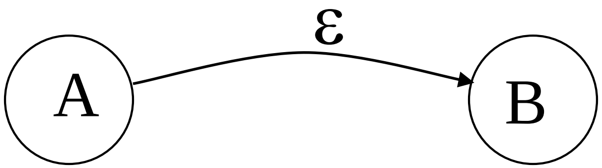
- Alphabet still { 0, 1 }



- The operation of the automaton is not completely defined by the input
  - On input “11” the automaton could be in either state

Epsilon Moves

- Another kind of transition: ε-moves



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

Deterministic and Nondeterministic Automata

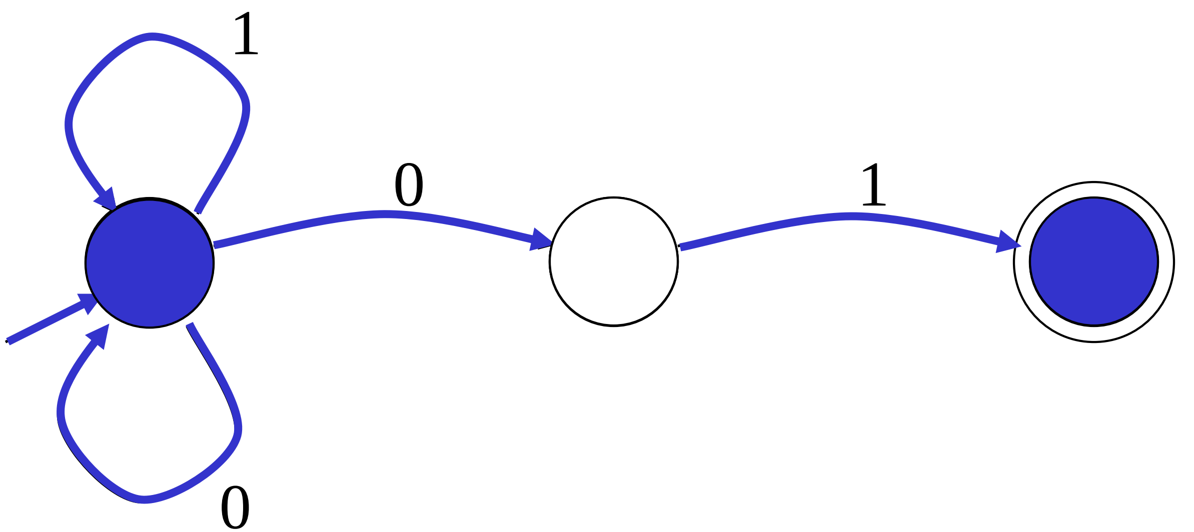
- Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No ε-moves
- Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have ε-moves
- *Finite* automata have *finite* memory
  - Need only to encode the current state

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 1
- Rule: NFA accepts if it <u>can</u> get in a final state

NFA vs. DFA (1)

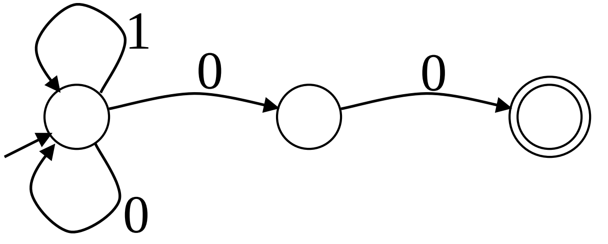
- NFAs and DFAs recognize the same set of languages (regular languages)

- DFAs are easier to implement
  - There are no choices to consider

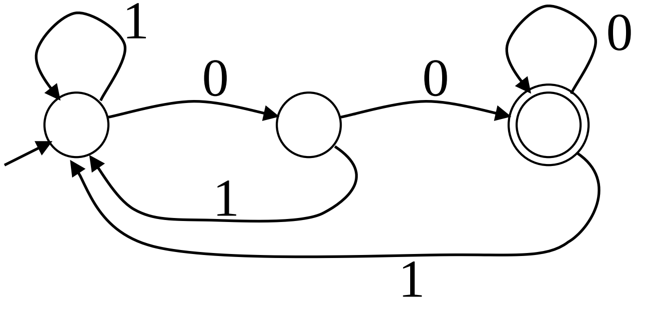
NFA vs. DFA (2)

- For a given language NFA can be simpler than DFA

NFA



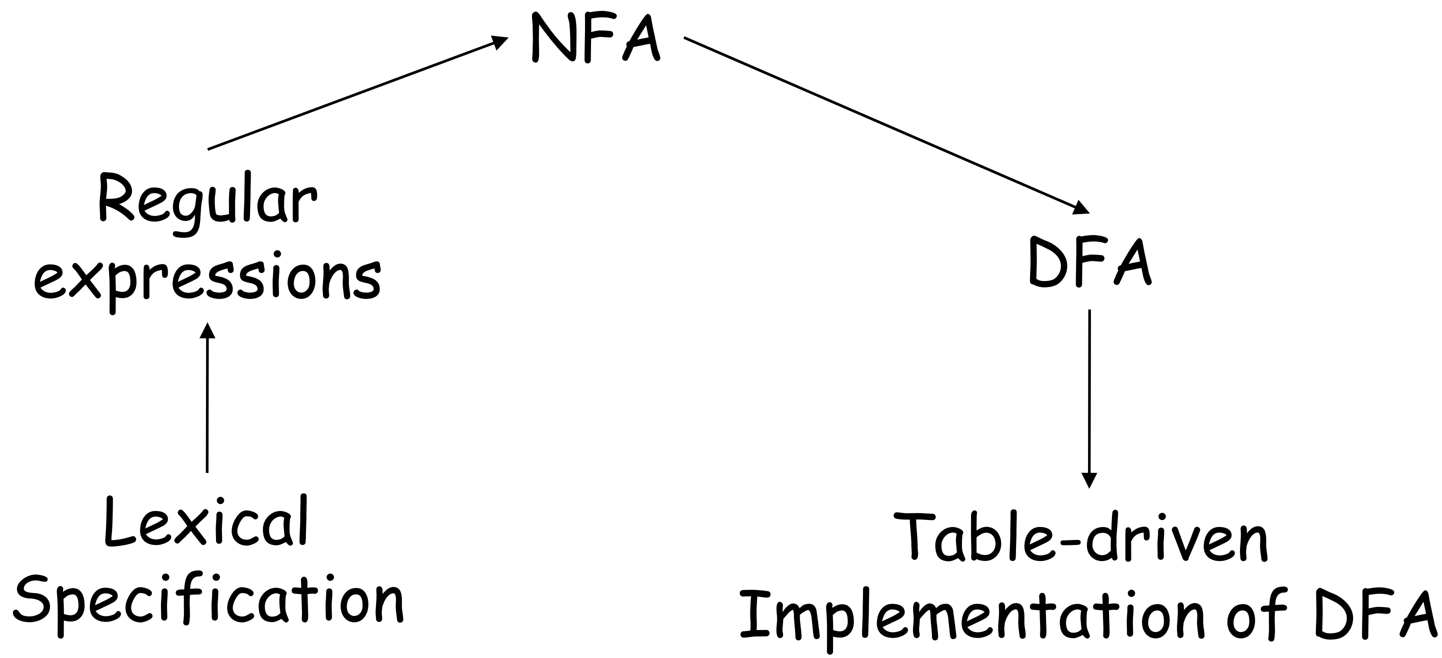
DFA



- DFA can be exponentially larger than NFA

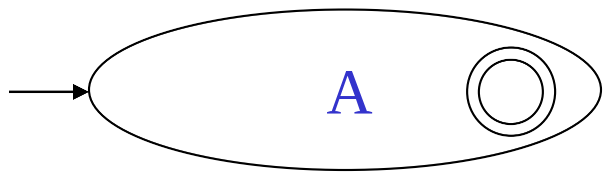
Regular Expressions to Finite Automata

- High-level sketch

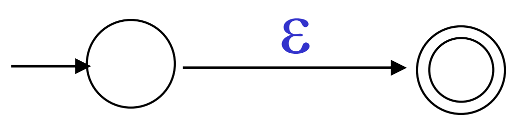


Regular Expressions to NFA (1)

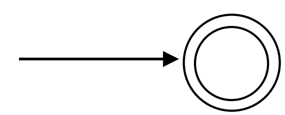
- For each kind of rexp, define an NFA
  - Notation: NFA for rexp *A*



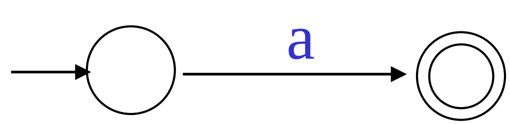
- For ε



or

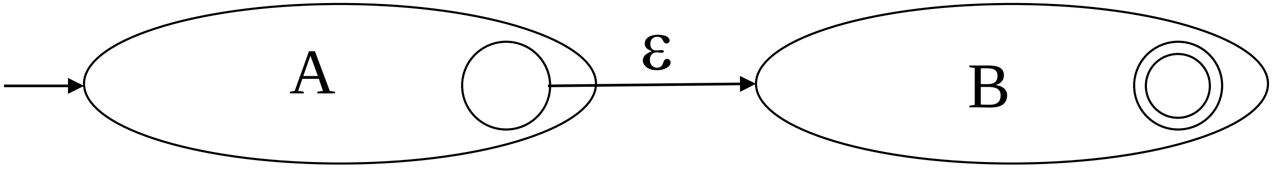


- For input a

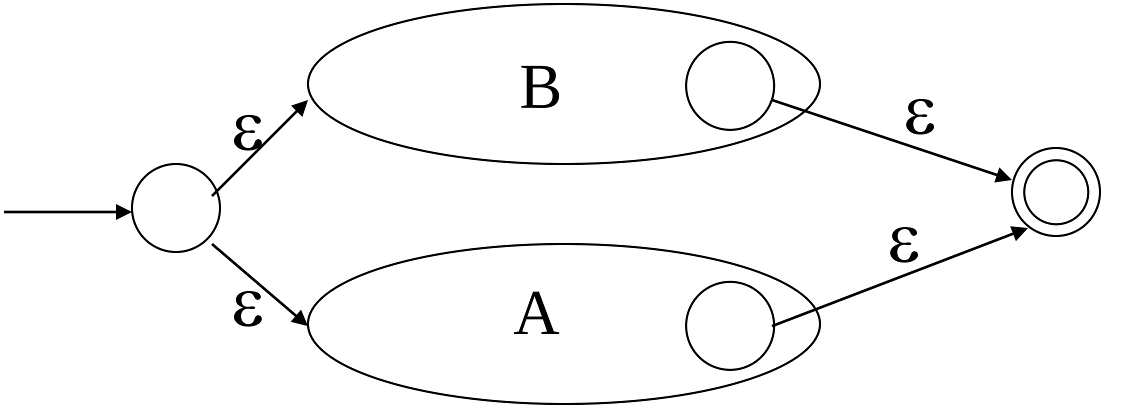


Regular Expressions to NFA (2)

- For AB

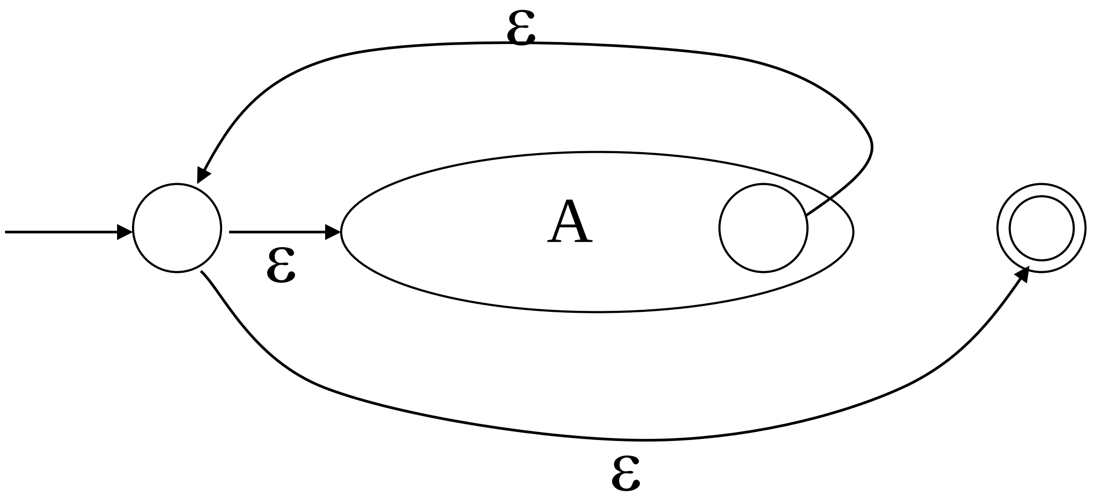


- For $A \mid B$



Regular Expressions to NFA (3)

- For *A**

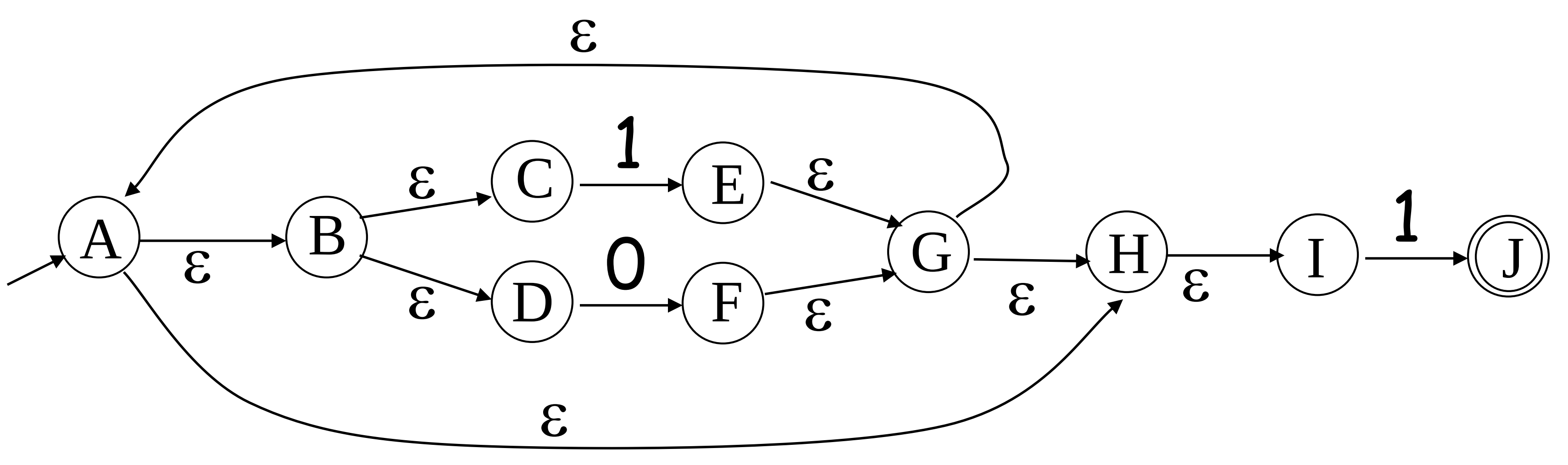


Example of RegExp -> NFA conversion

- Consider the regular expression

  (1 | 0)*1

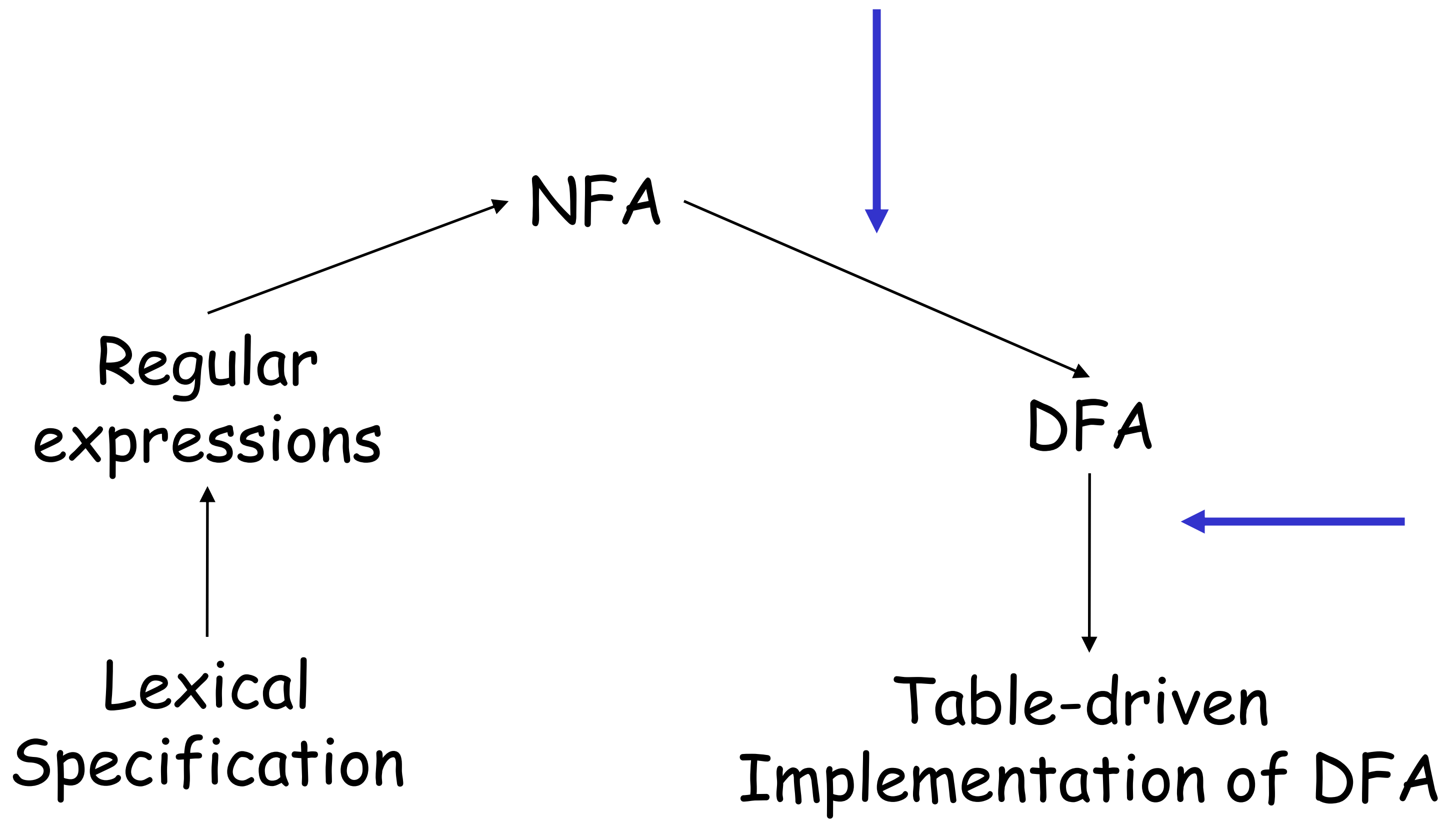
- The NFA is



A Side Note on the Construction

- To keep things simple, all the machines we built had exactly one final state.
- Also, we never merged ("overlapped") states when we combined machines.
  - E.g., we didn't merge the start states of the A and B machines to create the A|B machine, but created a new start state.
  - This avoided certain glitches: e.g., try A*|B*
- Resulting machines are very suboptimal: many extra states and ε transitions.
- But the DFA transformation gets rid of this excess, so it doesn't matter.

Next


NFA
Regular expressions
DFA
Lexical Specification
Table-driven Implementation of DFA

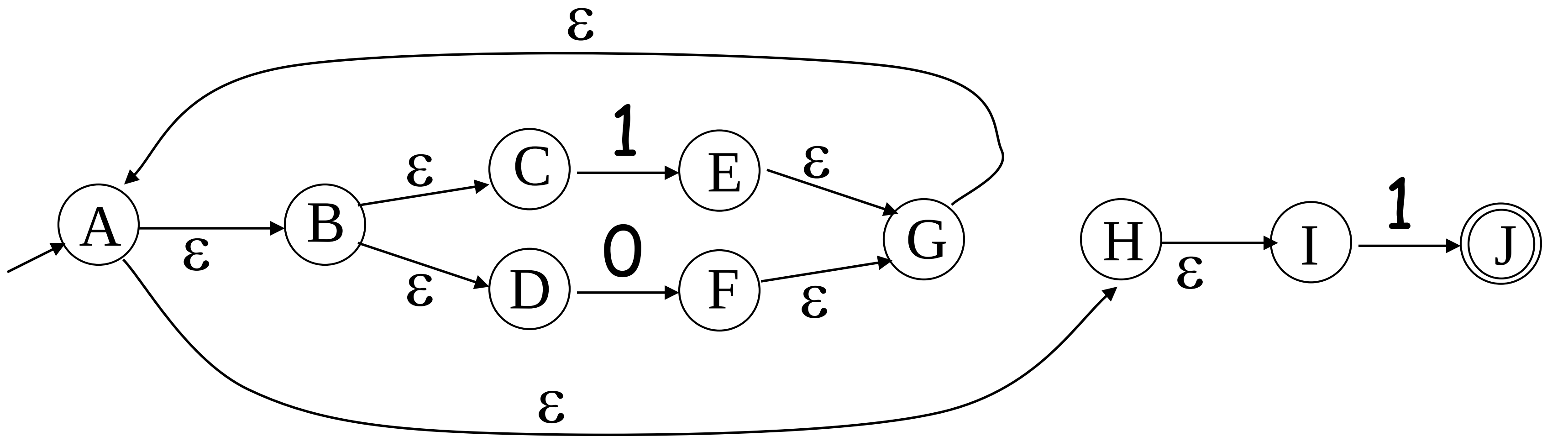
NFA to DFA: The Trick

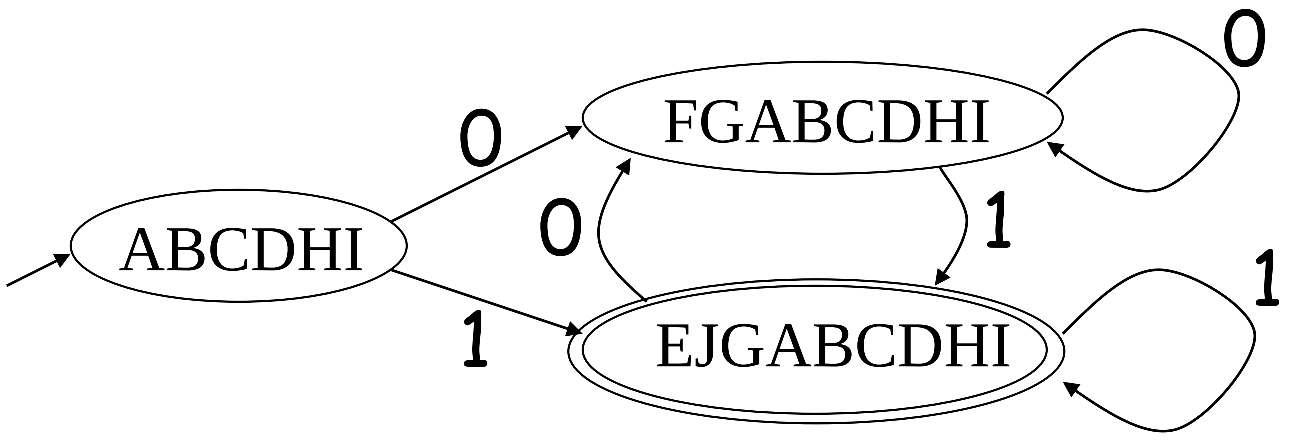
- Simulate the NFA
- Each state of resulting DFA
  = a non-empty subset of states of the NFA
- Start state
  = the set of NFA states reachable through ε-moves from NFA start state
- Add a transition $S \rightarrow^{a} S'$ to DFA iff
  - S' is the set of NFA states reachable from the states in S after seeing the input a
    - considering ε-moves as well

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 non-empty subsets are there?
  - $2^N - 1$ = finitely many, but exponentially many

NFA -> DFA Example

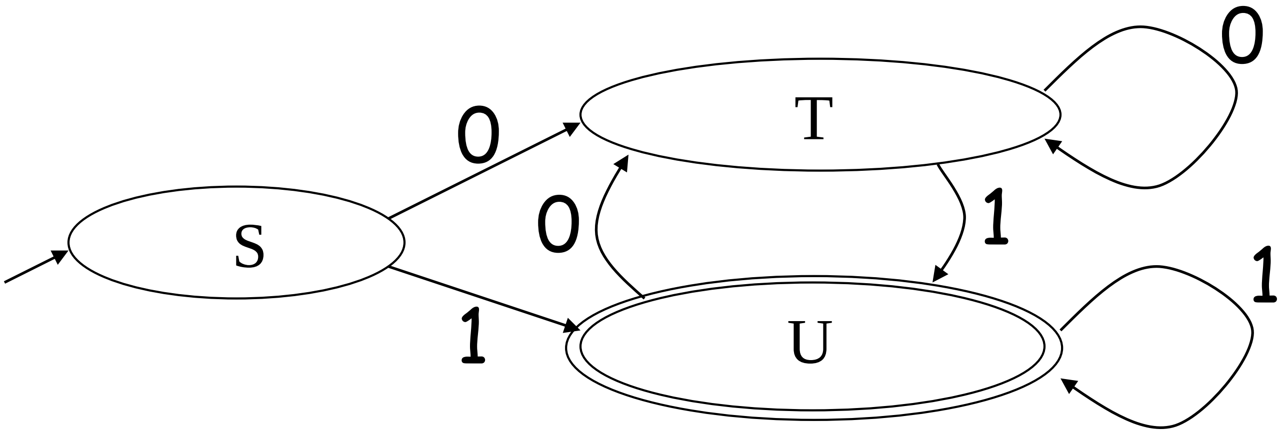




Implementation

- A DFA can be implemented by a 2D table T
  - One dimension is "states"
  - Other dimension is "input symbol"
  - For every transition $S_i \to^a S_k$ 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



	0	1
S	T	U
T	T	U
U	T	U

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

Assignments

- Draw DFAs for the following REs.
- (a) The set of all strings which start and end with the same digit.

  $1(0^*1)^*+0(1^*0)^*$
- (b) The set of all strings representing a binary number where the sum of its digits is even.

  $(0^*10^*1)^*0^*$
- (c) The set of all strings that contain the substring 10100.

  $(0+1)^*10100(0+1)^*$