CSC 310 - Programming Languages

Regular Languages

Lexical Analysis

- The goal of lexical analysis is to partition an input string into substrings where each substring is a token.
- Example:

```
if (i == j)
    z = 0;
else
    z = 1;
```

  is a string of characters:

```
if (i == j)\n\tz = 0;else\n\tz = 1;
```

- A lexical analyzer is called a lexer or a scanner

Tokens

- A *token* corresponds to a set of strings
- These sets depend on the programming language
- Examples:
  - Identifiers: strings of letters or digits starting with a digit
  - Integer: a non-empty string of digits
  - Keyword (reserved word): “if”, “else”, . . .
  - Whitespace: a non-empty sequence of spaces, newlines, and tabs

What are Tokens used for?

- Classify program substrings according to role
- The output of lexical analysis is a stream of tokens
- The input to the parser is a stream of tokens
- The parser relies on token distinctions, for example, an identifier is treated differently than a keyword

Regular Languages

- There are several formalisms for specifying tokens
- Regular languages are the most popular
  - Simple and useful theory
  - Easy to understand
  - Efficient implementations

Languages

- **Definition.** Let Σ be a set of characters. A language over Σ is a set of strings of characters drawn from Σ. Σ is called the alphabet.

Examples of Languages

- Natural language
  - Alphabet: English characters
  - Language: English sentences
  - Note: not every string of English characters is an English sentence
- Programming language
  - Alphabet: ASCII
  - Language: C programs
  - Note: The ASCII character set is different from the English character set

Regular Expressions

- The lexical structure of most programming languages can be specified with regular expressions.
- *Languages* are sets of strings - we need some notation for specifying which sets we want, that is, which strings are in the set.
- A *regular expression* (RE) is a notation for a regular language
- If A is a regular expression, then we write $L(A)$ to refer to the language denoted by A.

Fundamental Regular Expressions

A	$L(A)$	Notes
a	$\{\mathrm{a}\}$	singleton set for each symbol 'a' in the alphabet Σ
ϵ	$\{\epsilon\}$	empty string
$\varnothing$	$\{\ \}$	empty language

- These are the basic building blocks of regular expressions.

Operations on Regular Expressions

A	$L(A)$	Notes
rs	$L(r)L(s)$	concatenation – r followed by s
$r\|s$	$L(r) \cup L(s)$	combination (union) – r or s
$r*$	$L(r)*$	zero or more occurrences of r (Kleene closure)

- Precedence: $*$ (highest), concatenation, $|$ (lowest)
- Parenthesis can be used to group REs as needed
- We abbreviate 'i' 'f' as 'if' (concatenation)

Examples

- $L(\text{if} \mid \text{then} \mid \text{else}) = \{\text{“if”}, \text{“then”}, \text{“else”}\}$
- $L((0 \mid 1)\ (0 \mid 1)) = \{\text{“00”}, \text{“01”}, \text{“10”}, \text{“11”}\}$
- $L(0^*) = \{\text{“”}, \text{“0”}, \text{“00”}, \text{“000”}, \ldots\}$
- $L((1|0)(1|0)^*) =$ set of binary numbers with possible leading zeros

Abbreviations

Abbreviation	Meaning	Notes
$r+$	$(rr*)$	one or more occurrences
$r?$	$(r\|\epsilon)$	zero or one occurrence
$[a-z]$	$(a\|b\|\ldots\|z)$	one character in given range
$[abxyz]$	$(a\|b\|x\|y\|z)$	one of the given characters
$[\hat{}abc]$	$\overline{[abc]}$	any character except the given characters

- The basic operations generate all possible regular expressions, but common abbreviations are used for convenience.

Regular Languages and Finite Automata

- Result from formal language theory: regular expressions and finite automata both define the class of regular languages
- Thus, lexical analysis uses:
  - Regular expressions for specification
  - Finite automata for implementation (automatic generation of lexical analyzers)

Finite Automata

- A finite automata is a *recognizer* for the set of strings of a regular language
- A finite automaton consists of:
  - A finite input alphabet Σ
  - A set of states S
  - A start state n
  - A set of accepting states $F \subseteq S$
  - A set of transitions in $S \rightarrow S$ (mappings from states to states)

Finite Automata

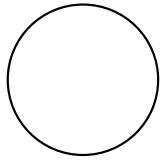
- Transition notation

  $s_1 \rightarrow^a s_2$

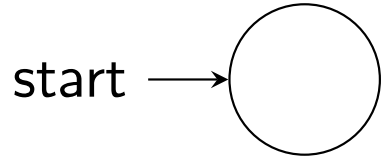
  is read: in state s_1 on input a go to state s_2
- Each transition "consumes" a character from the input
- At the end of input (or no transition possible)
  - If in accepting state, accept ($s \in L(R)$)
  - Otherwise, reject ($s \notin L(R)$)

Finite Automata State Graphs

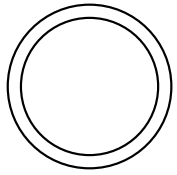
- A state:



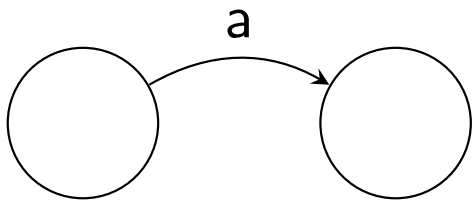
- A start state:



- An accepting state:

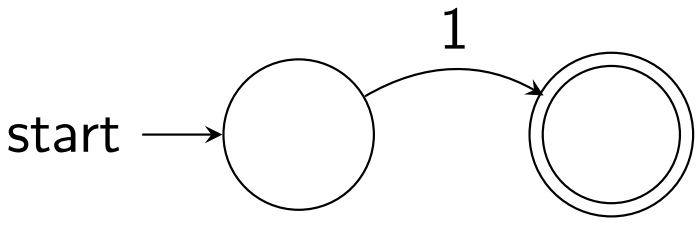


- A transition:



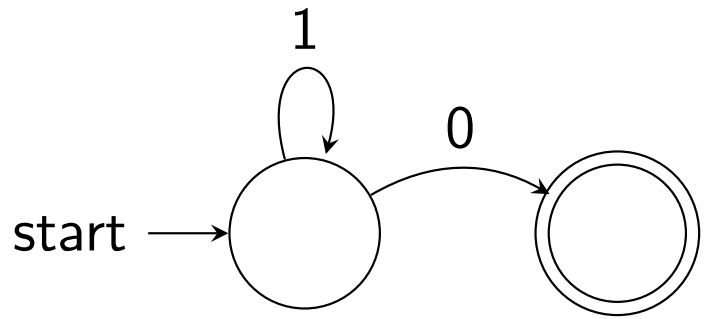
A Simple Example

- A finite automaton that accepts only "1";



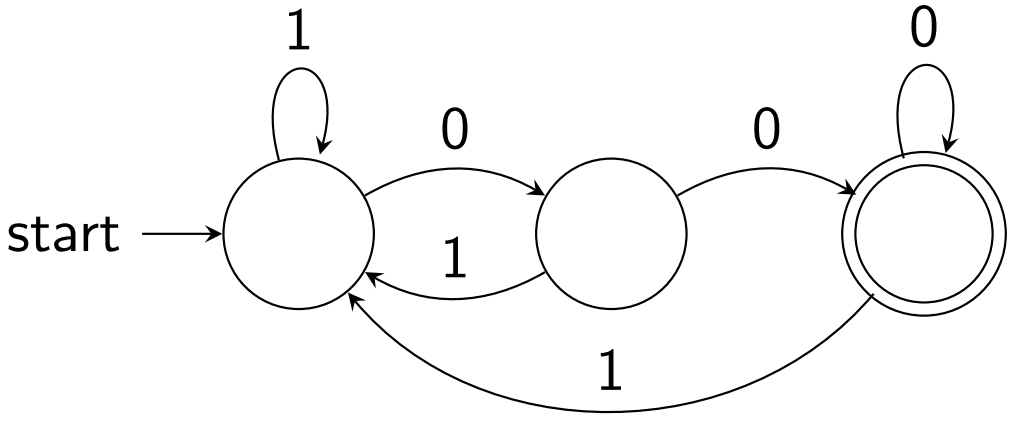
Another Simple Example

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



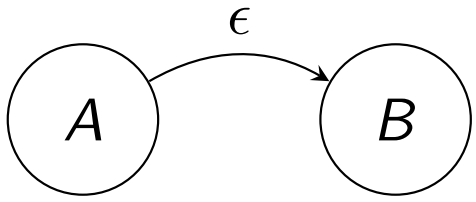
Another Example

- Alphabet: $\{0, 1\}$



Epsilon Transitions

- Epsilon transitions:



- The automaton can move from state A to state B without consuming input

Deterministic and Non-Deterministic Automata

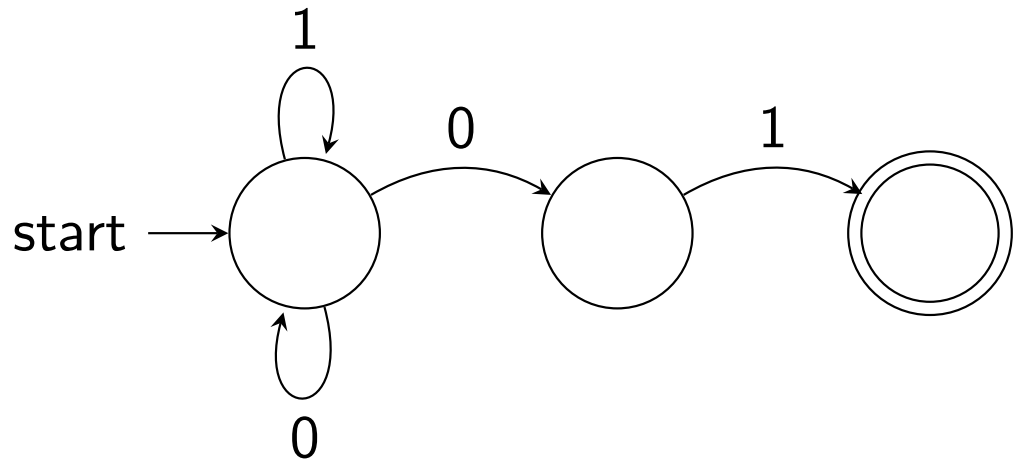
- Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No epsilon transitions
- Non-deterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have epsilon transitions
- Finite automata have finite memory – only enough to encode 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 epsilon transitions
  - which of multiple transitions for a single input to take

Acceptance of NFAs

- An NFA can get into multiple states



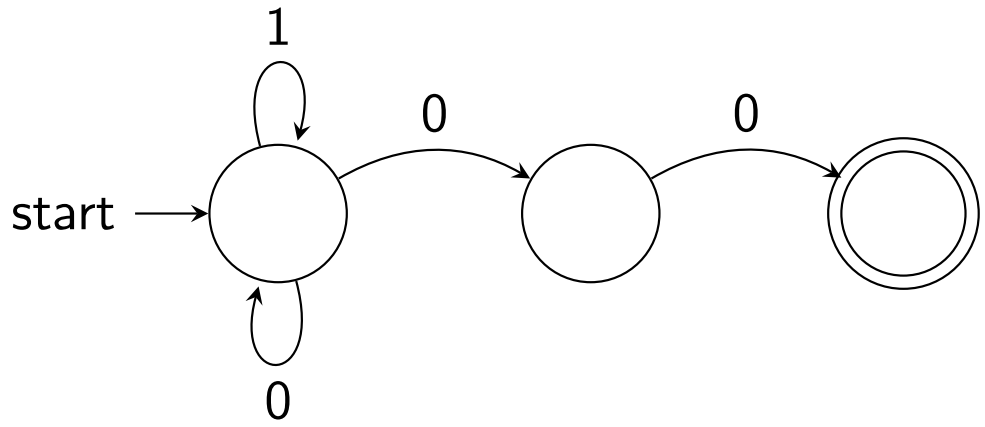
- An NFA accepts an input if it can get in a final state
- Exampe input: 1 0 1

NFA versus DFA

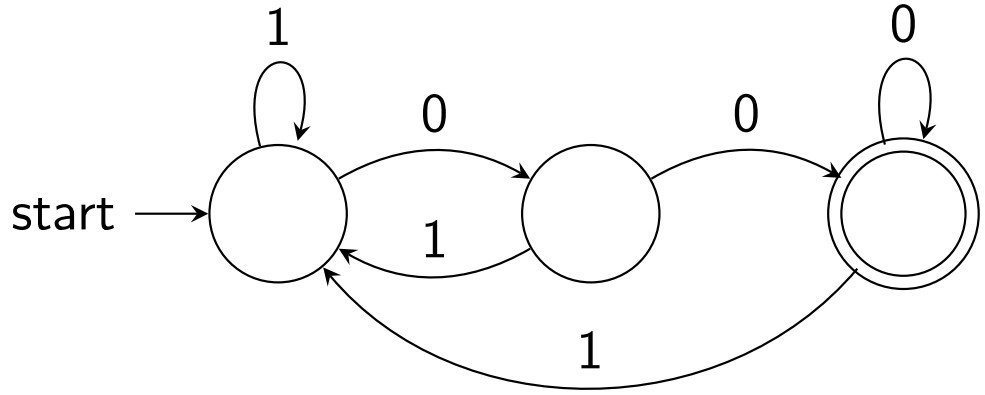
- NFAs and DFAs recognize the same set of languages (regular languages)
- DFAs are easier to implement
- A DFA can be exponentially larger than an equivalent NFA

NFA versus DFA

- For a given language the NFA can be simpler than the DFA
  - NFA:



  - DFA:

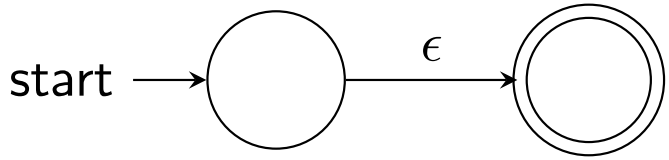


Regular Expressions to Finite Automata

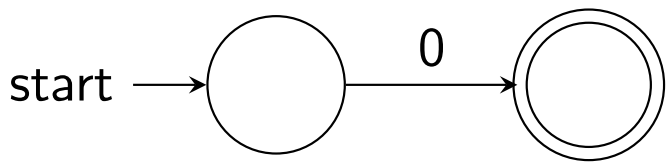
- The implementation of a lexical specification as a finite automata has the following transformations:
  1. Lexical specification
  2. Regular expressions
  3. NFA
  4. DFA
  5. Table driven implementation of DFA

Regular Expressions to NFA

- We can define an NFA for each basic regular expression and than connect the NFAs together based on the operators
- Basic regular expressions
  - ϵ transition

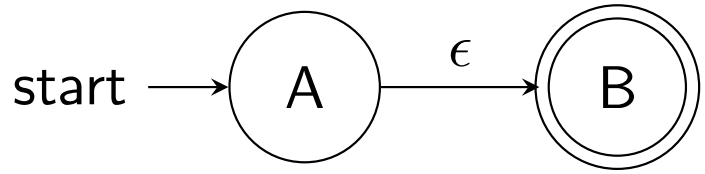


  - Input charater '0'

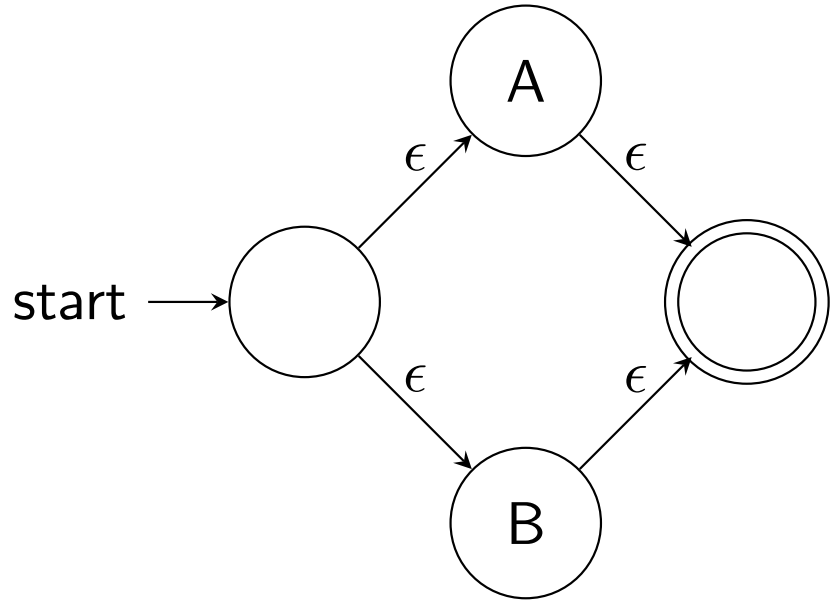


Regular Expressions to NFA

- AB: make an ϵ transition from the accepting state of A to start state of B

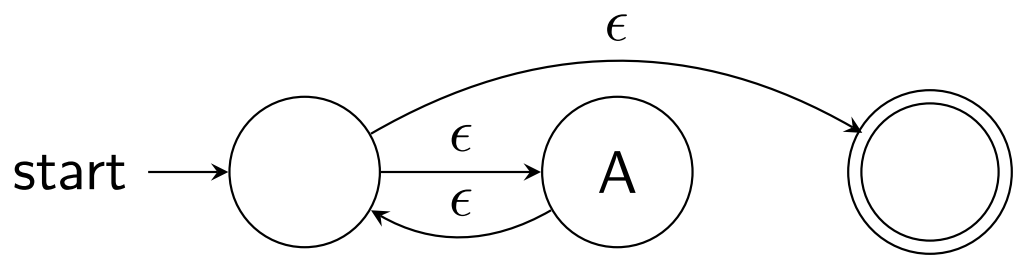


- $A|B$: create a new start state and add ϵ transitions from the new start state to the start states of A and B, then create a new accepting state and add ϵ transitions from the accepting states of A and B to the new accepting state



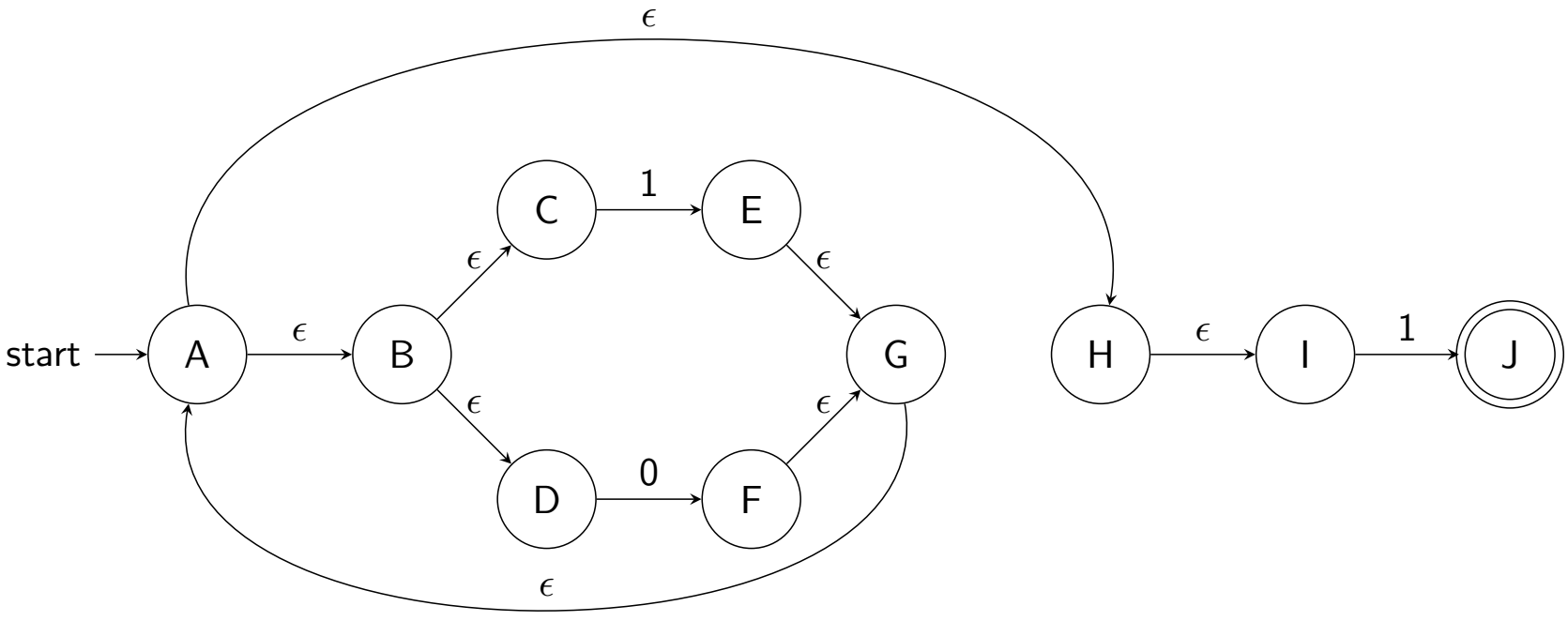
Regular Expressions to NFA

- $A*$: create a new start state and accepting state and add an ϵ transitions: from the new start state to the start state of A, from the accepting state of A to the new start state, and from the new start state to the new accepting state.



Regular Expressions to NFA Example

- Consider the regular expression: `(1|0)*1`
- The NFA is



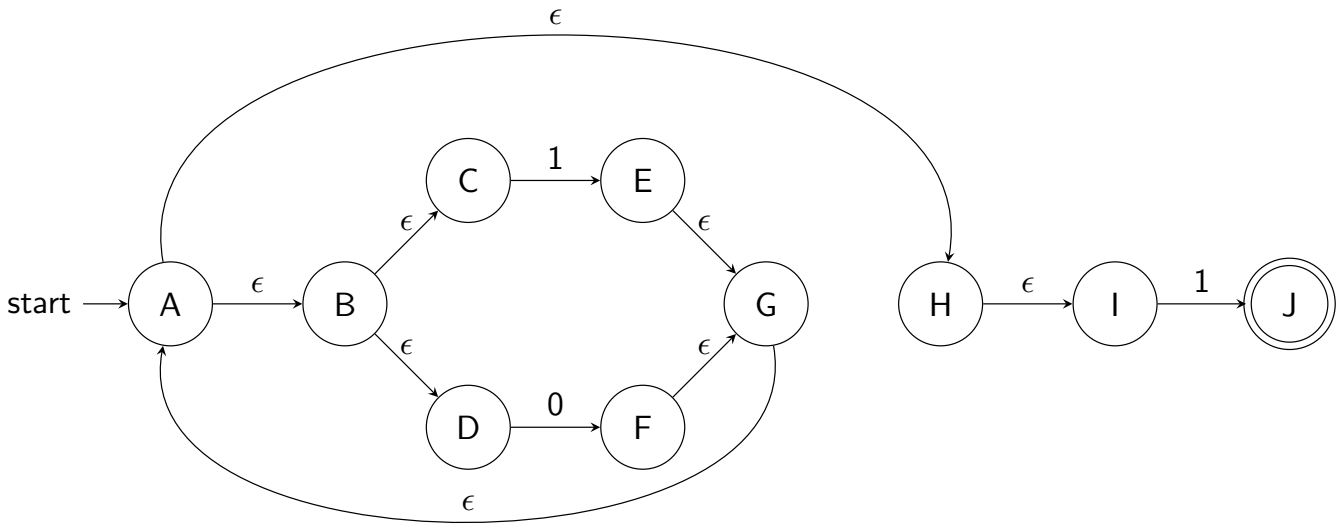
NFA to DFA (The Trick)

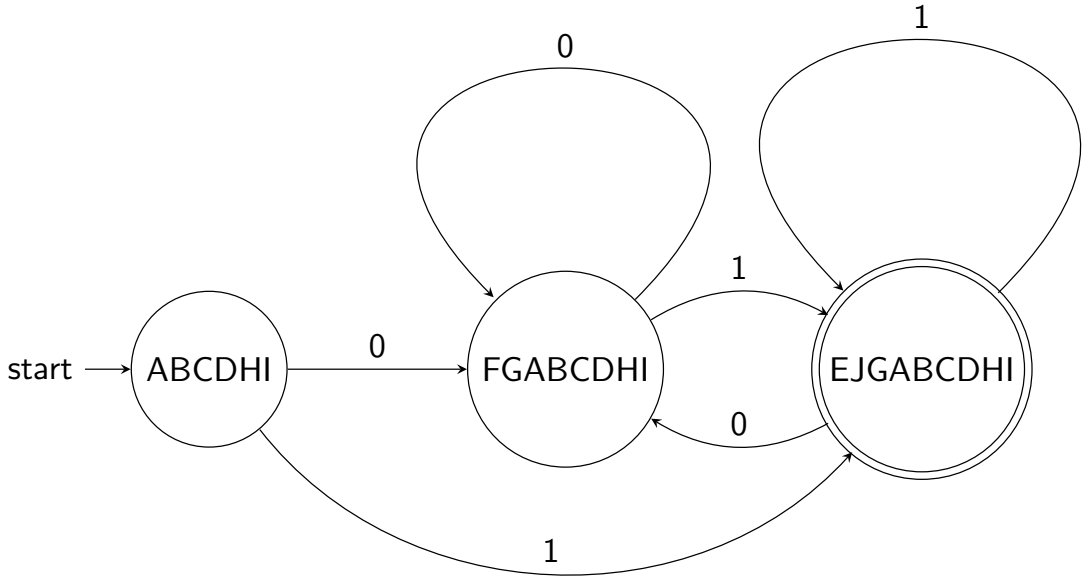
- Simulate the NFA
- Each state of the DFA is a non-empty subset of states of the NFA
- The start state is the set of NFA states reachable through epsilon transitions from the NFA start state
- Add a transition $S \rightarrow^a S'$ to the DFA if and only if S' is the set of NFA states reachable from any state in S after seeing the input a (considering epsilon transitions as well)

NFA to DFA Remark

- An NFA may be in many states at any time
- If there are N states, the NFA must be in some subset of those N states
- There are $2^N - 1$ possible subsets (finitely many)

NFA to DFA Example

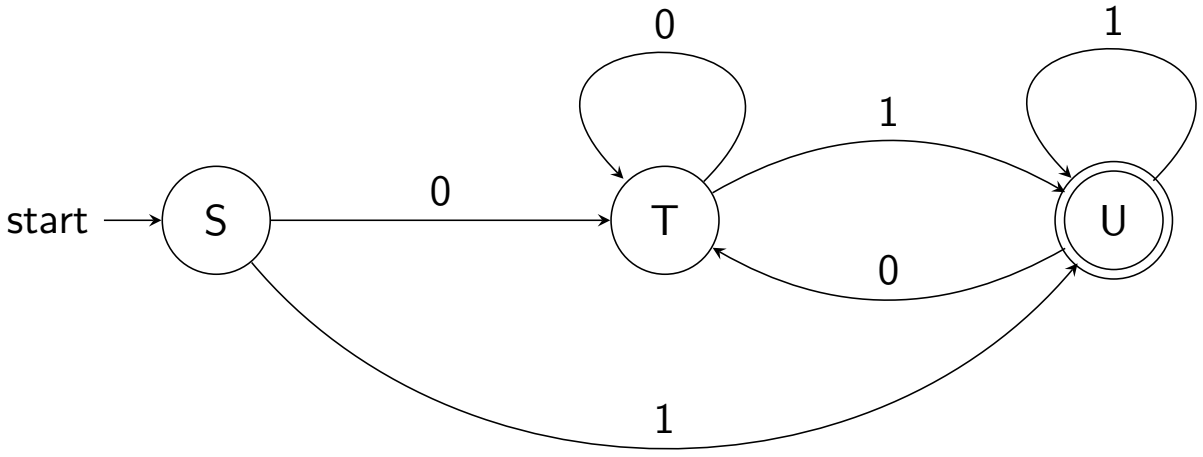




Implementation

- A DFA can be implemented by a 2D table T
  - One dimension is "states"
  - The other dimension is "input symbols"
  - For every transition $S_i \rightarrow^a S_k$ define $T[i, a] = k$
- DFA "execution"
  - If in state S_i and input a, then read $T[i, a]k$ and skip to state S_k
  - This is efficient

Example: Table Implementation of a DFA



	0	1
S	T	U
T	T	U
U	T	U

Implementation Continued

- The NFA to DFA conversion is the core operation of lexical analysis tools such as lex
- But, DFAs can be huge
- In practice, lex-like tools trade off speed for space in the choice of NFA and DFA representations

Theory versus Practice

- DFAs *recognize* lexemes. A lexer must return a *type of acceptance* (token type) rather than simply an accept/reject indication
- DFAs consume the complete string and accept or reject it. A lexer must *find* the end of the lexeme in the input stream and then find the *next* one, etc.