

Formal and Natural Languages

Natural languages are the languages that people speak, such as English, Spanish, and French. They were not designed by people (although people try to impose some order on them); they evolved naturally.

Formal languages are languages that are designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Chemists use a formal language to represent the chemical structure of molecules. And most importantly:

Programming languages are formal languages that have been designed to express computations.

Ambiguity - Natural languages are full of ambiguity, which people deal with by using contextual clues and other information. Formal languages are designed to be nearly or completely unambiguous, which means that any statement has exactly one meaning, regardless of context.

Redundancy - In order to make up for ambiguity and reduce misunderstandings, natural languages employ lots of redundancy. As a result, they are often verbose. Formal languages are less redundant and more concise.

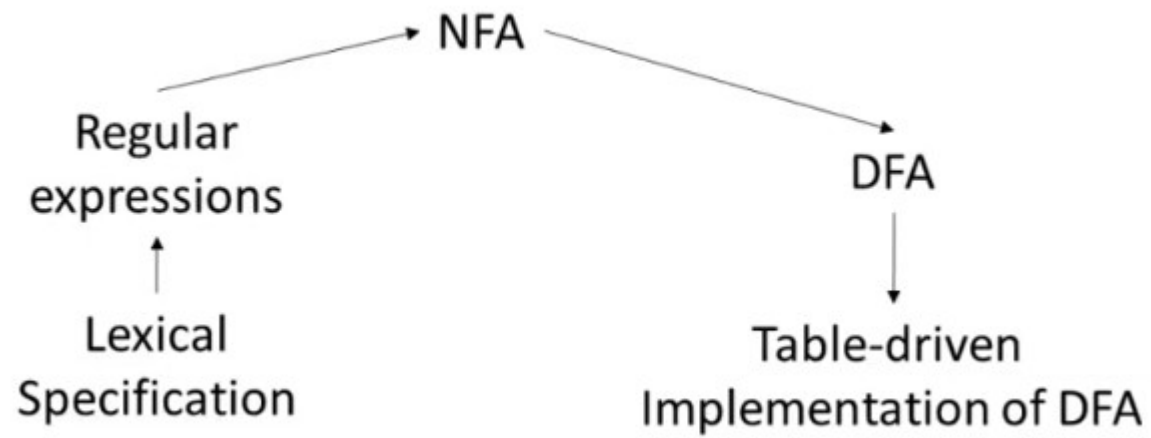
Literalness - Formal languages mean exactly what they say. On the other hand, natural languages are full of idiom and metaphor. If someone says, "The other shoe fell", there is probably no shoe and nothing falling.

Formal languages tend to have strict rules about syntax. For example, $3+3=6$ is a syntactically correct mathematical statement, but $3=+6\$$ is not. H_2O is a syntactically correct chemical name, but $_2Zz$ is not.

Syntax rules come in two flavours, pertaining to tokens and structure. Tokens are the basic elements of the language, such as words, numbers, and chemical elements. One of the problems with $3=+6\$$ is that $\$$ is not a legal token in mathematics (at least as far as we know). Similarly, $_2Zz$ is not legal because there is no element with the abbreviation Zz .

The second type of syntax rule pertains to the structure of a statement— that is, the way the tokens are arranged. The statement $3=+6\$$ is structurally illegal because you can't place a plus sign immediately after an equal sign. Similarly, molecular formulas have to have subscripts after the element name, not before.

When you read a sentence in English or a statement in a formal language, you have to figure out what the structure of the sentence is (although in a natural language you do this subconsciously). This process is called parsing.



Formal Language Theory

In mathematics, computer science, and linguistics, a formal language consists of words whose letters are taken from an alphabet and are well-formed according to a specific set of rules.

A formal language L over an alphabet Σ is a subset of Σ^* , that is, a set of words over that alphabet.

Regular Expressions

Regular expressions are used to define patterns of characters; they are used in UNIX tools such as awk, grep, vi and, of course, lex.


A regular expression is just a form of notation, used for describing sets of words. For any given set of characters in a set Σ , a regular expression over Σ is defined by:

The empty string, ε , which denotes a string of length zero, $\{\varepsilon\}$ and means take nothing from the input. It is most commonly used in conjunction with other regular expressions to denote optionality.

Any character in Σ may be used in a regular expression. For instance, if we write 'a' as a regular expression, this means take the letter a from the input; ie. it denotes the (singleton) set of words $\{a\}$

two base cases





1/ The union operator, $|$, which denotes the union of two sets of words. Thus the regular expression $a|b$ denotes the set $\{“a”, “b”\}$, and means take either the letter a or the letter b from the input.

2/ Writing two regular expressions side-by-side is known as concatenation; thus the regular expression ab denotes the set $\{“ab”\}$ and means take the character a followed by the character b from the input.

3/ The Kleene closure of a regular expression, denoted by $*$, indicates zero or more occurrences of that expression. Thus a^* is the (infinite) set $\{\epsilon, "a", "aa", "aaa", \dots\}$ and means take zero or more 'a' from the input.

Brackets may be used in a regular expression to enforce precedence or increase clarity.

The above are the three compound expressions

Regular Expressions

Each *regular expression* represents a set of strings.

- **Symbol:** For each symbol a in the language, the regular expression a denotes the string a .
- **Alternation:** If M and N are 2 regular expressions, then $M|N$ denotes a string in M or a string in N .
- **Concatenation:** If M and N are 2 regular expressions, then MN denotes a string $\alpha\beta$ where α is in M and β is in N .
- **Epsilon:** The regular expression ϵ denotes the empty string.
- **Repetition:** The *Kleene closure* of M , denoted M^* is the set of zero or more concatenations of M .

Kleene closure binds tighter than concatenation, and concatenation binds tighter than alternation.

Regular Expression Examples

$(0 1)^*$	$\epsilon, 0, 1, 10, 11, 100, 101, 110, \dots$
$(0 1)^*0$	$0, 10, 100, 110, 1000, 1010, \dots$
$(a b)^*$	$\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, \dots$
$(a b)^*aa(a b)^*$	a string of a's and b's that contains at least one pair of consecutive a's.
$b^*(abb^*)^*(a \epsilon)$	a string of a's and b's with no consecutive a's.
$ab^*(c \epsilon)$	a string starting with an a, followed by zero or more b's and ending in an optional c. $a, ac, ab, abc, abb, abbc, \dots$

Regular Expression Shorthands

The following abbreviations are generally used:

- $[axby]$ means $(a|x|b|y)$
- $[a - e]$ means $[abcde]$
- $M?$ means $(M|\epsilon)$
- M^+ means (MM^*)
- $.$ means any single character except a newline character
- " a^{+*} " is a quotation and the string in quotes literally stands for itself.

Disambiguation Rules for Scanners

Is do99 an identifier or a keyword (do) followed by a number (99)?

Most modern lexical-analyser generators follow 2 rules to disambiguate situations like above.

- Longest match: The longest initial sub-string that can match any regular expression is taken as the next token.
- Rule priority: In the case where the longest initial sub-string is matched by multiple regular expressions, the first regular expression that matches determines the token type.

So do99 is an identifier.

Finite Automata

Regular expressions are good for specifying lexical tokens. Finite automata are good for recognising regular expressions.

A finite automaton consists of a set of nodes and edges. Edges go from one node to another node and are labelled with a symbol. Nodes represent states. One of the nodes represents the start node and some of the nodes are final states.

A deterministic finite automaton (DFA) is a finite automaton in which no pairs of edges leading away from a node are labelled with the same symbol.

A nondeterministic finite automaton (NFA) is a finite automaton in which two or more edges leading away from a node are labelled with the same symbol.

A finite automaton consists of

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

- Transition

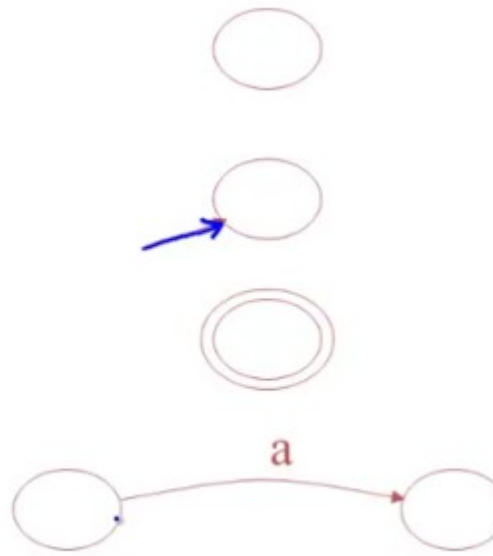
$$s_1 \xrightarrow{a} s_2$$

- Is read

In state s_1 on input a go to state s_2

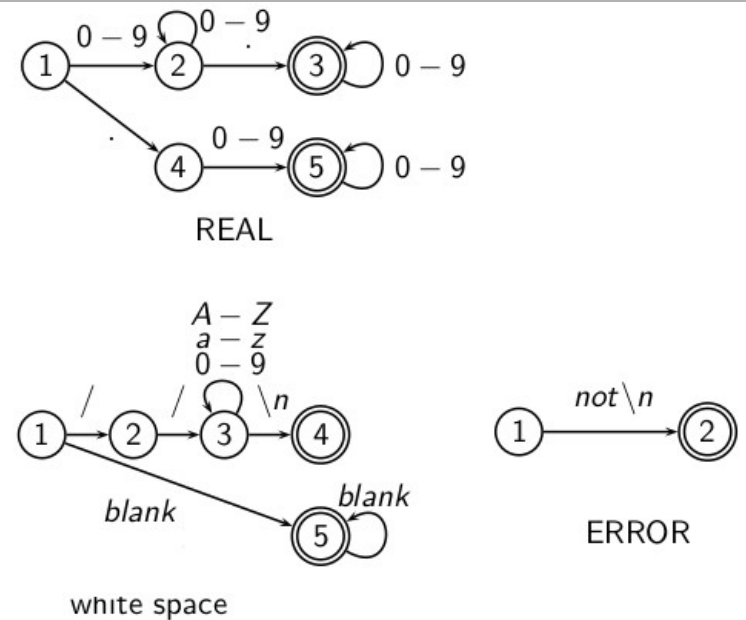
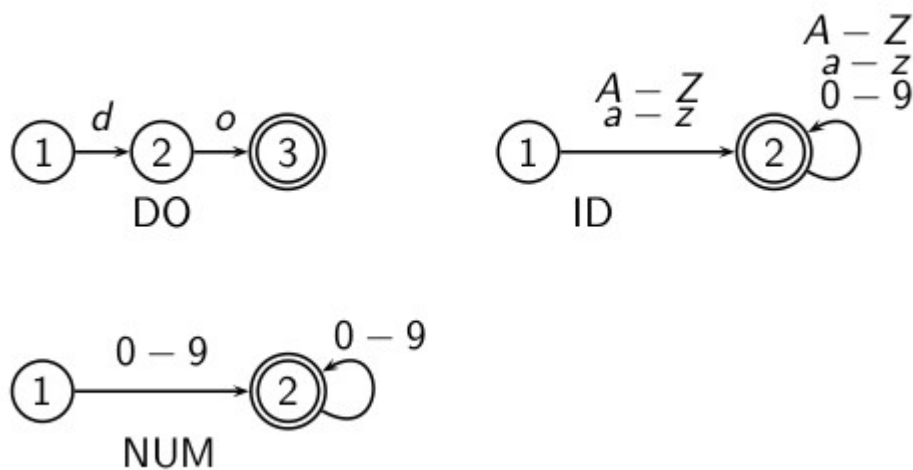
- If end of input and in accepting state \Rightarrow accept
- Otherwise \Rightarrow reject

- A state
- The start state
- An accepting state
- A transition

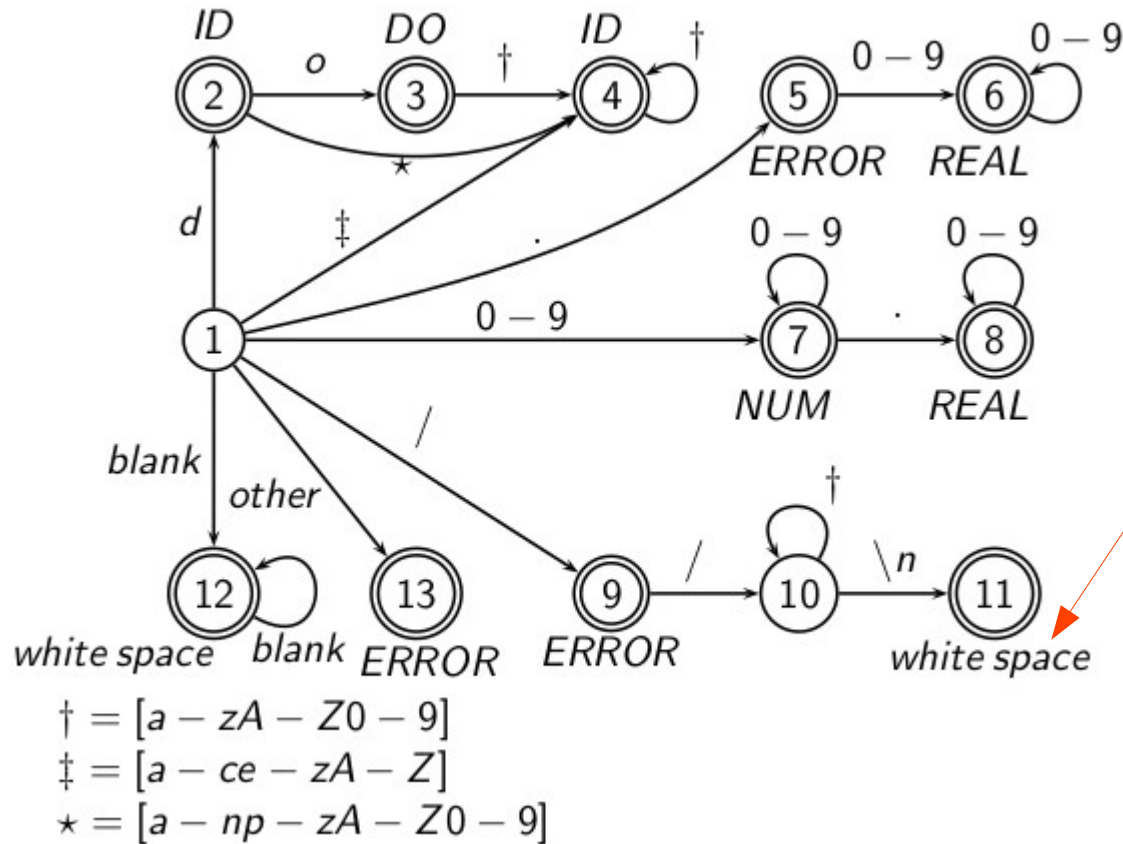


Regular Expressions for Tokens

do	DO
$[a-zA-Z][a-zA-Z0-9]^*$	ID
$[0-9]^+$	NUM
$([0-9]^+ "." [0-9]^+) ([0-9]^+ "." [0-9]^+)$	REAL
$" / " [a-zA-Z0-9]^* \backslash n (" " \backslash n " \backslash t ")^+$	comment or white space



Combined Finite Automaton



Each final state must be labelled with the token-type that it accepts

Encoding a Finite Automaton

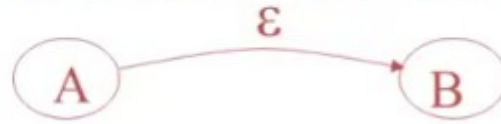
A finite automaton can be encoded by:

- **Transition matrix:** a 2-dimensional array, indexed by input character and state number, that contains the next state.

```
int edges[] [] = { /* ws,..., 0, 1, 2, ... d, e, f, ... o, ... */
    /* state 0 */ { 0,..., 0, 0, 0, ..., 0, 0, 0, ..., 0, ... },
    /* state 1 */ { 0,..., 7, 7, 7, ..., 2, 4, 4, ..., 4, ... },
    /* state 2 */ { 0,..., 4, 4, 4, ..., 4, 4, 4, ..., 3, ... },
    /* state 3 */ { 0,..., 4, 4, 4, ..., 4, 4, 4, ..., 4, ... },
    /* state 4 */ { 0,..., 4, 4, 4, ..., 4, 4, 4, ..., 4, ... },
    /* state 5 */ { 0,..., 6, 6, 6, ..., 0, 0, 0, ..., 0, ... },
    ...
}
```


- **action array:** an array, indexed by final state number, that contains the resulting action, e.g. if the final state is 2 then return ID, if the final state is 3 then return DO, etc.

- Another kind of transition: ϵ -moves



move to another state
without consuming input

- 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

- 
- NFAs and DFAs recognize the same set of languages
 - regular languages
 - DFAs are faster to execute
 - There are no choices to consider
 - NFAs are, in general, smaller

How an NFA operates

We begin in the start state (usually labelled 0) and read the first character on the input.

Each time we are in a state, reading a character from the input, we examine the outgoing transitions for this state, and look for one labelled with the current character. We then use this to move to a new state. There may be more than one possible transition, in which case we choose one at random.

If at any stage there is an output transition labelled with the empty string, ϵ , we may take it without consuming any input. We keep going like this until we have no more input, or until we have reached one of the final states.

If we are in a final state, with no input left, then we have succeeded in recognising a pattern.

Otherwise we must backtrack to the last state in which we had to choose between two or more transitions, and try selecting a different one.

Basically, in order to match a pattern, we are trying to find a sequence of transitions that will take us from the start state to one of the finish states, consuming all of the input.

The key concept here is that: every NFA corresponds to a regular expression

Moreover, it is fairly easy to convert a regular expression to a corresponding NFA. To see how NFAs correspond to regular expressions, let us describe a conversion algorithm.

Thompson's construction algorithm, also called the McNaughton-Yamada-Thompson algorithm is a method of transforming a regular expression into an equivalent nondeterministic finite automaton

– Notation: NFA for rexp **M**



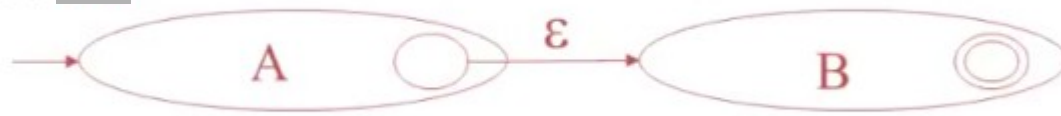
• For ϵ



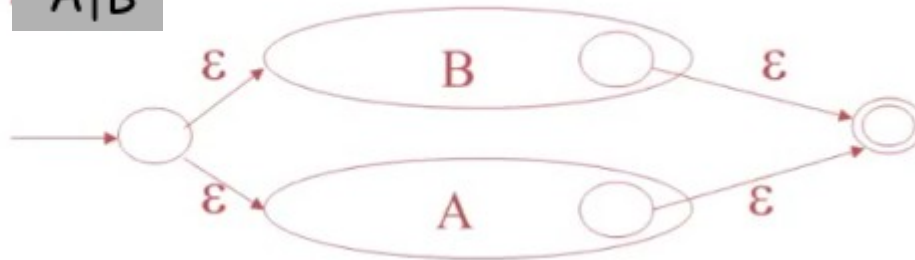
• For input **a**



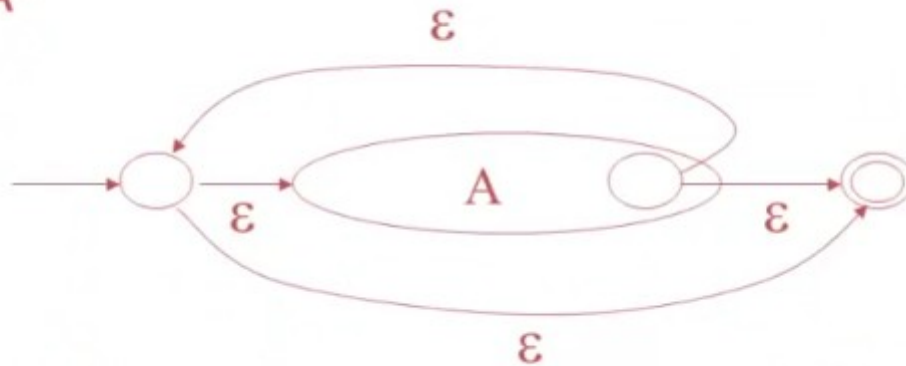
- For AB



- For $A|B$



- For A^*



With these rules, using the empty expression and symbol rules as base cases, it is possible to prove with mathematical induction that any regular expression may be converted into an equivalent NFA

- Consider the regular expression

$(1|0)^*1$



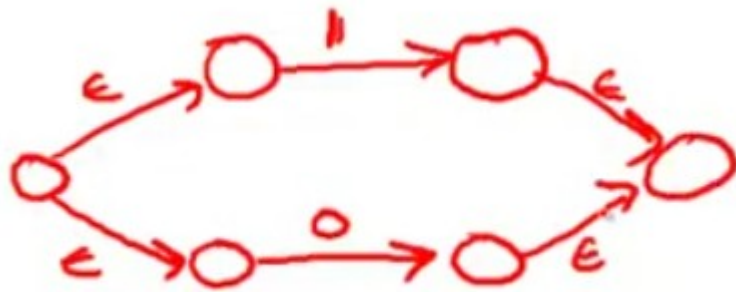
1



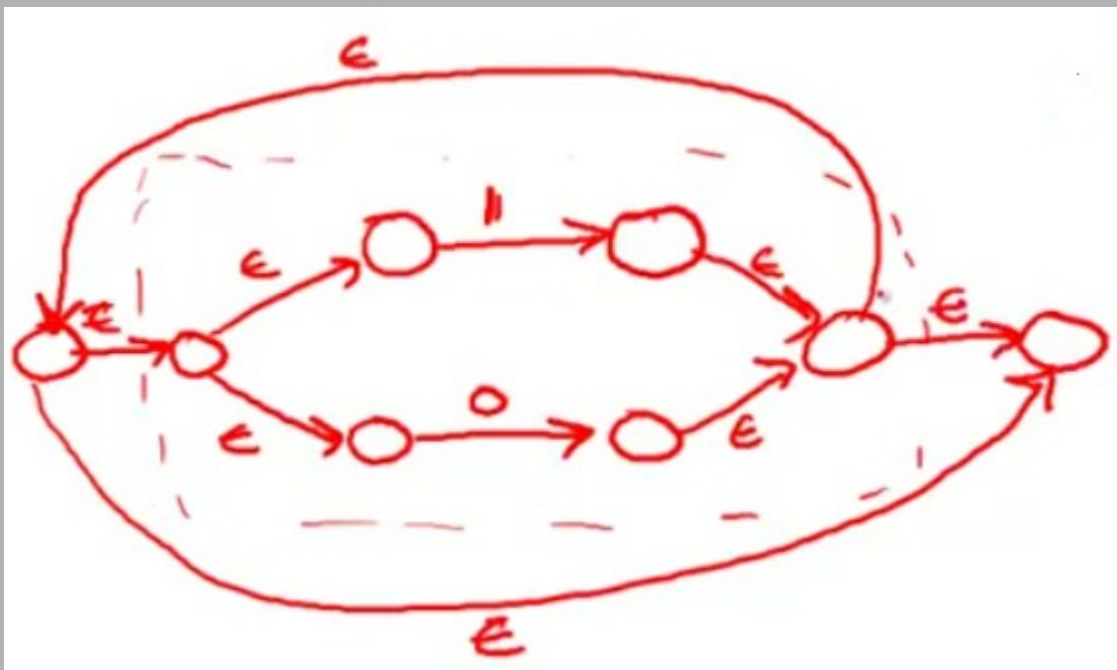
1



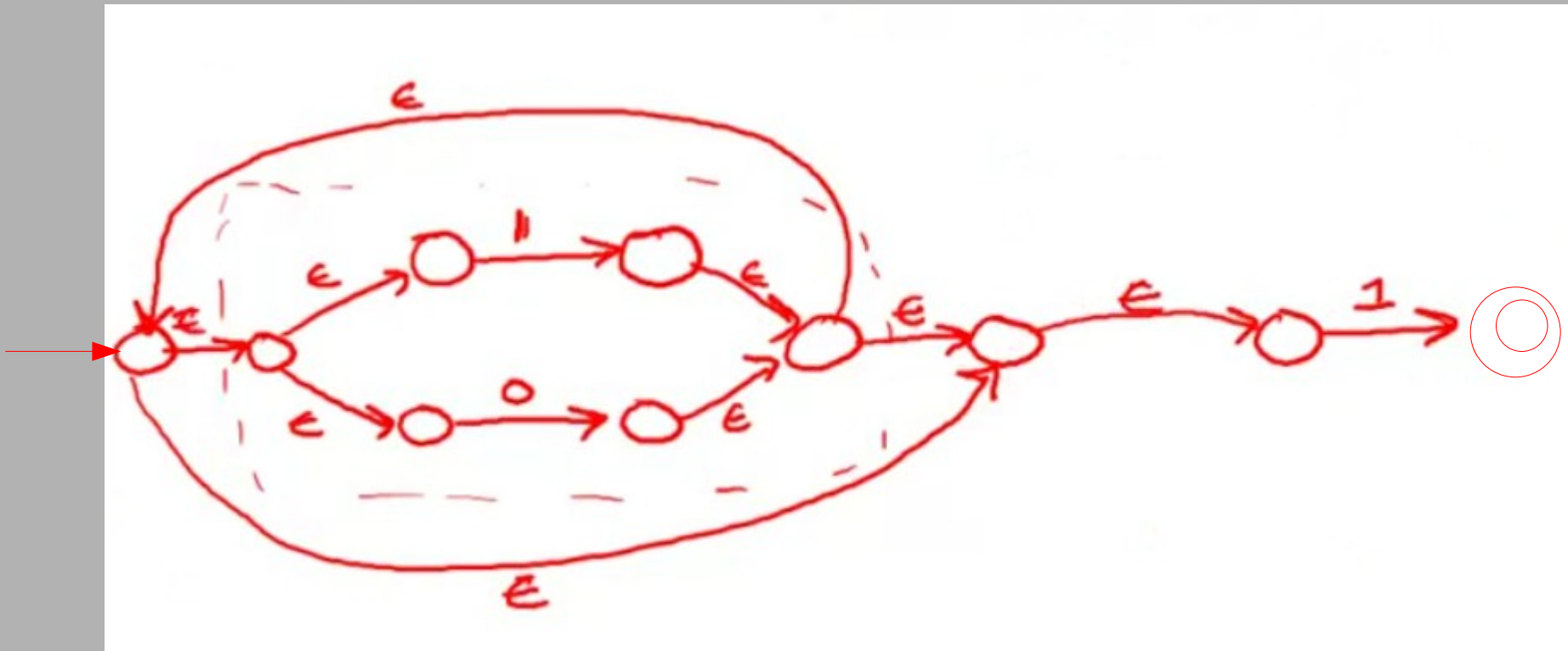
0



(1|0)



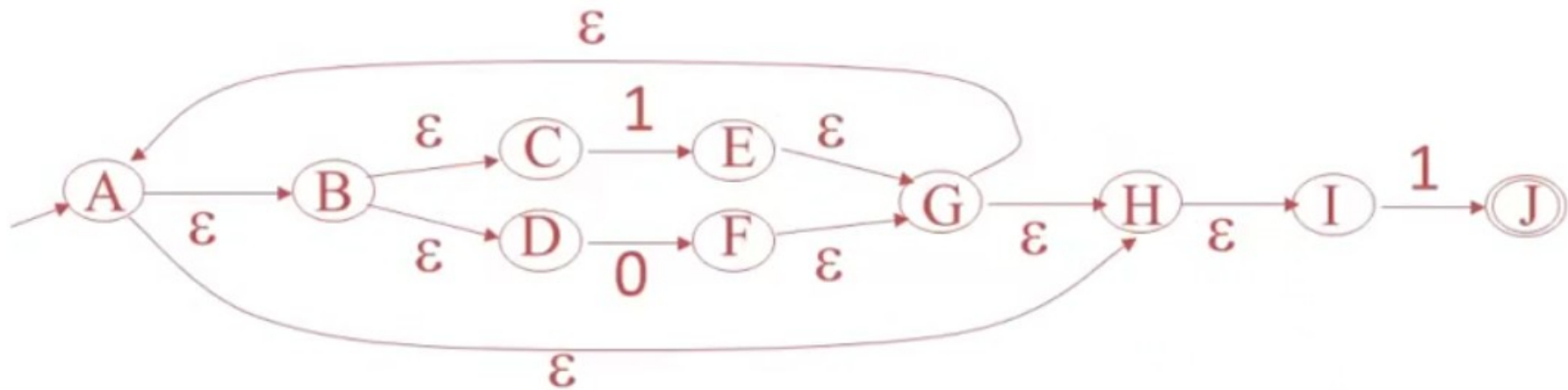
$(1|0)^*$



$(1|0)^*1$

- Consider the regular expression

$(1|0)^*1$




Deterministic Finite-State Automaton

NFAs are useful, in that they are easily constructed from regular expressions, and give us some kind of computational idea of how a scanner might work. However, since they involve making decisions, and backtracking if that decision was wrong, they are not really suitable for implementation using conventional programming languages.

Instead, we use a Deterministic Finite-State Automaton (DFA) which is a special case of a NFA with the additional requirements that:

There are no transitions involving ϵ

No state has two outgoing transitions based on the same symbol



Thus, when we are in some state in a DFA, there is no choice to be made; the operation of a DFA can very easily be converted into a program.

It is vital to note that every NFA can be converted to an equivalent DFA


We will define an algorithm to do this, for arbitrary NFAs the basic idea here is that sets of states in the NFA will correspond to just one state in the DFA.

Subset Construction - Constructing a DFA from an NFA

We merge together NFA states by looking at them from the point of view of the input characters.

From the point of view of the input, any two states that are connected by an ϵ -transition may as well be the same, since we can move from one to the other without consuming any character. Thus states which are connected by an ϵ -transition will be represented by the same states in the DFA.

If it is possible to have multiple transitions based on the same symbol, then we can regard a transition on a symbol as moving from a state to a set of states (ie. the union of all those states reachable by a transition on the current symbol). Thus these states will be combined into a single DFA state.



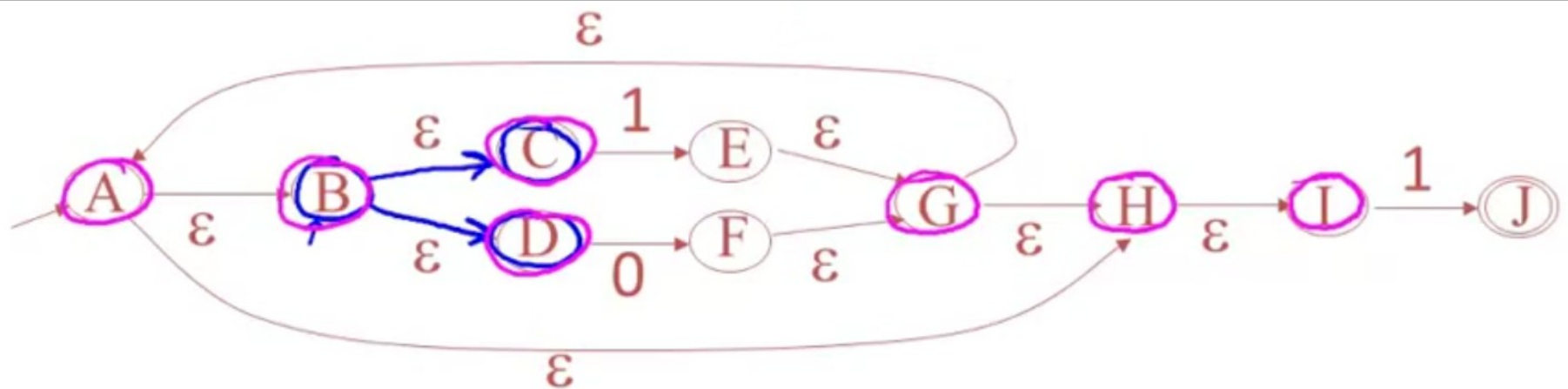
To perform this operation, let us define two functions:

The ϵ -closure function takes a state and returns the set of states reachable from it based on (one or more) ϵ -transitions. Note that this will always include the state itself. We should be able to get from a state to any state in its ϵ -closure without consuming any input.

The function move takes a state and a character, and returns the set of states reachable by one transition on this character.

$$\epsilon\text{-closure}(G) = \{A, B, C, D, G, H, I\}$$

$$\epsilon\text{-closure}(B) = \{B, C, D\}$$

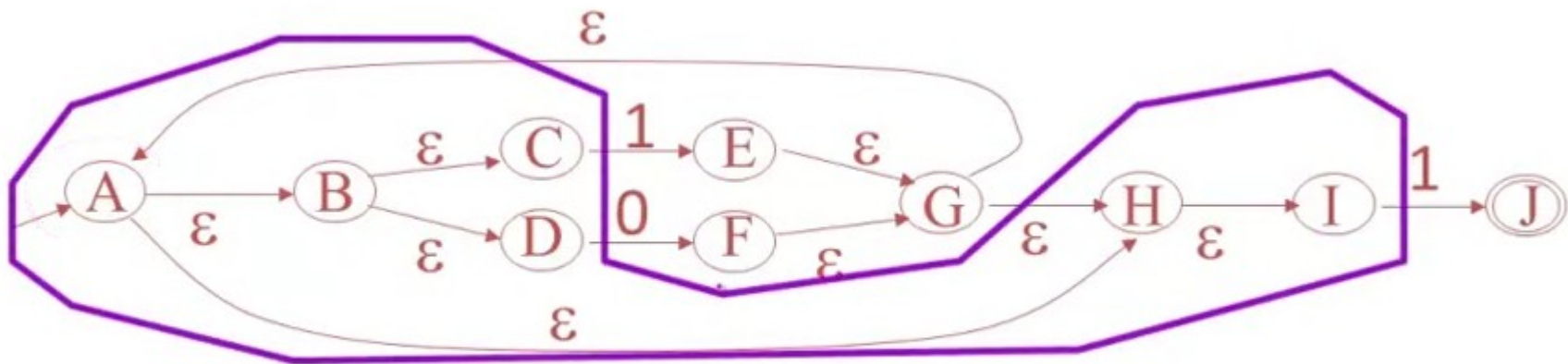


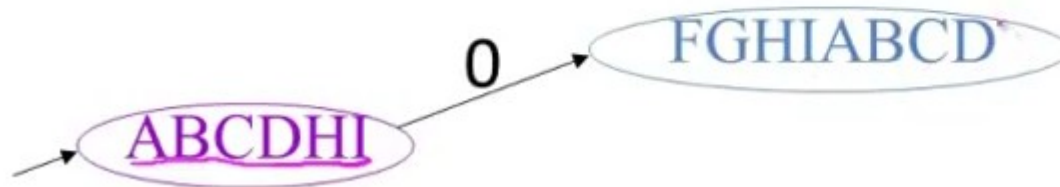
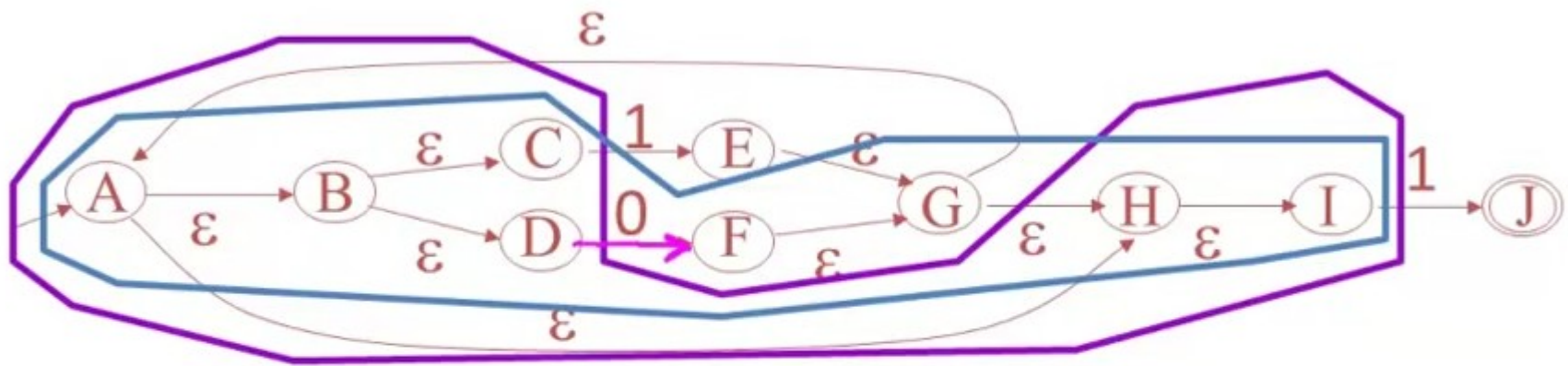
The Subset Construction Algorithm

- 1/ Create the start state of the DFA by taking the ϵ -closure of the start state of the NFA.
- 2/ Perform the following for the new DFA state:
 - For each possible input symbol:
 - 1/ Apply move to the newly-created state and the input symbol.
this will return a set of states.
 - 2/ Apply the ϵ -closure to this set of states, possibly resulting in a new set.
This set of NFA states will be a single state in the DFA.
- 3/ Each time we generate a new DFA state, we must apply step 2 to it. The process is complete when applying step 2 does not yield any new states.
- 4/ The finish states of the DFA are those which contain any of the finish states of the NFA.

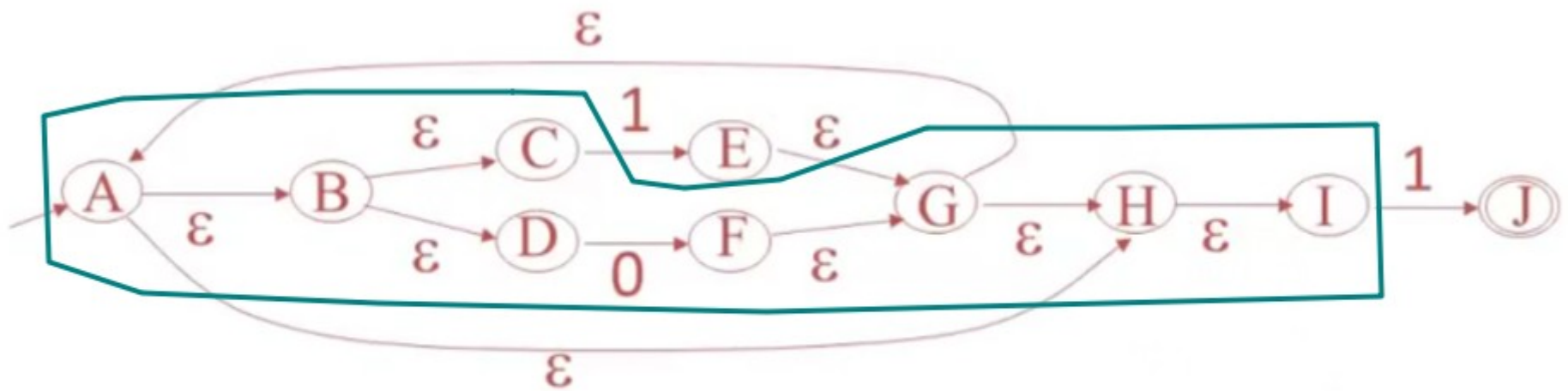
- Consider the regular expression

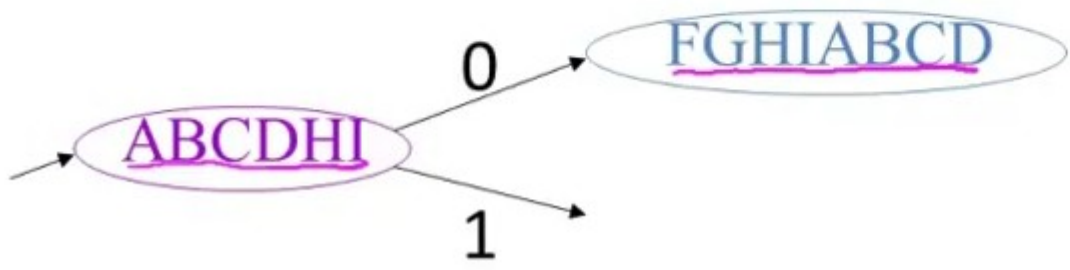
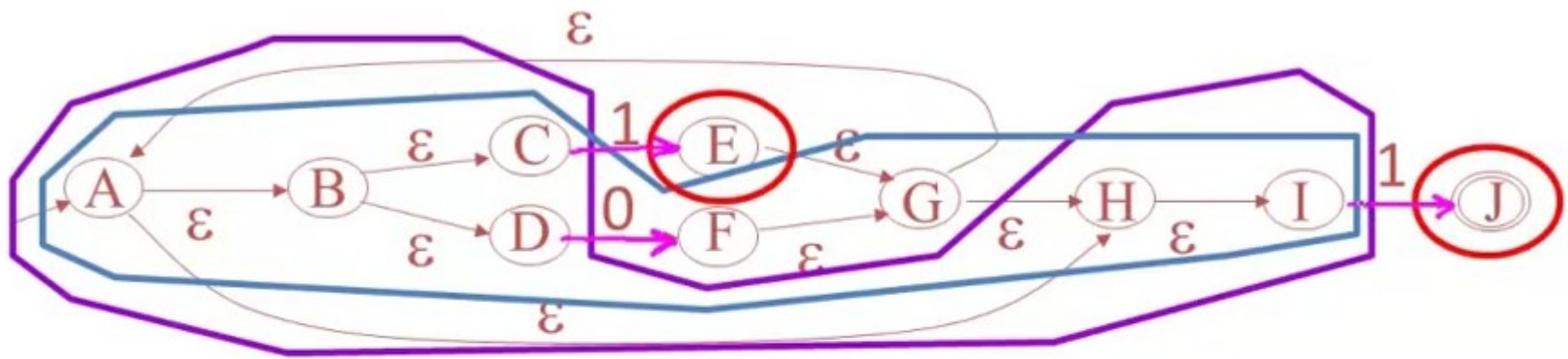
$(1|0)^*1$



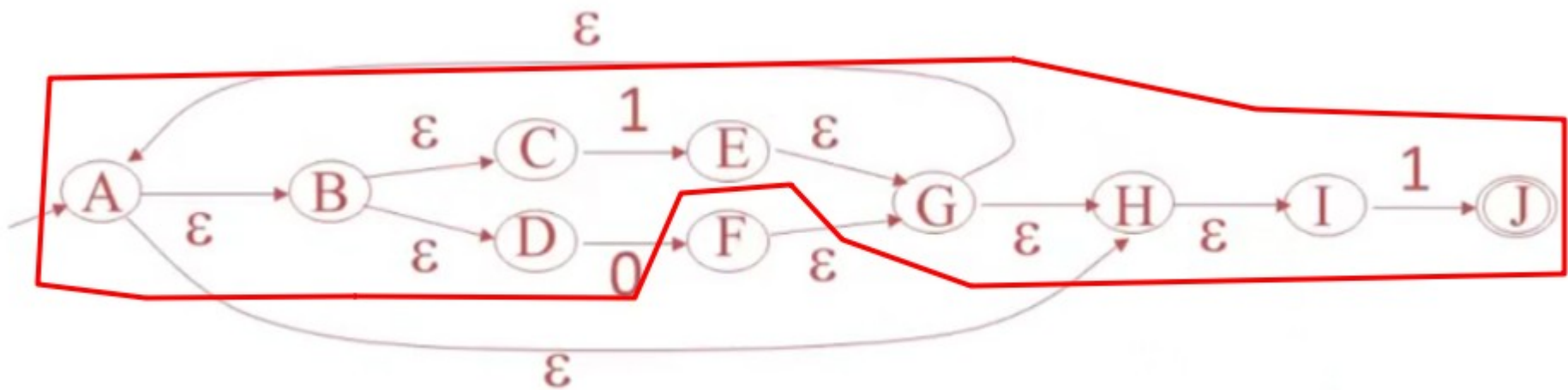


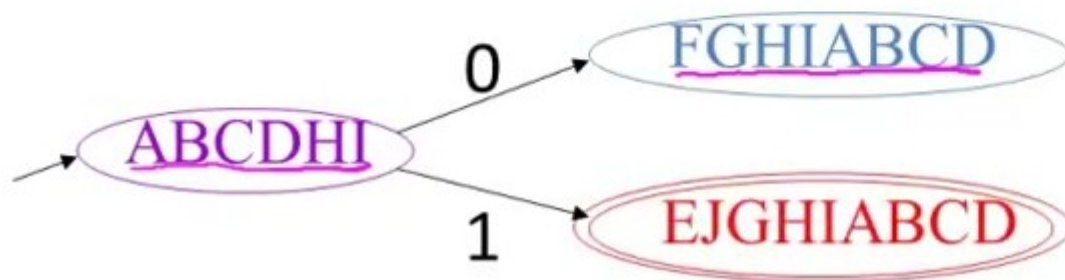
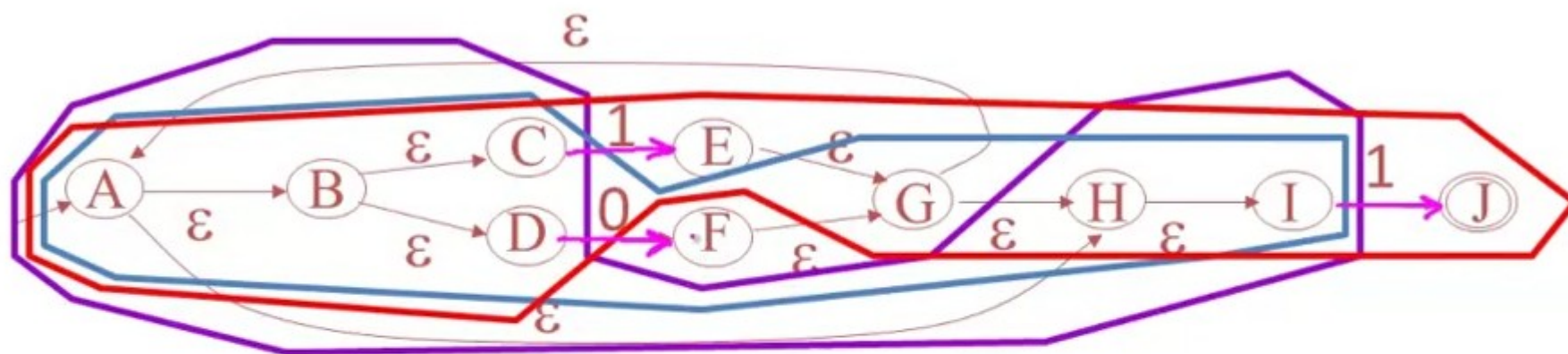
Epsilon closure of F

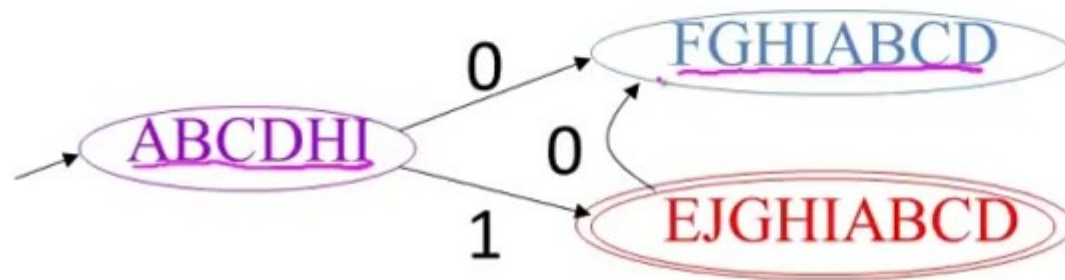
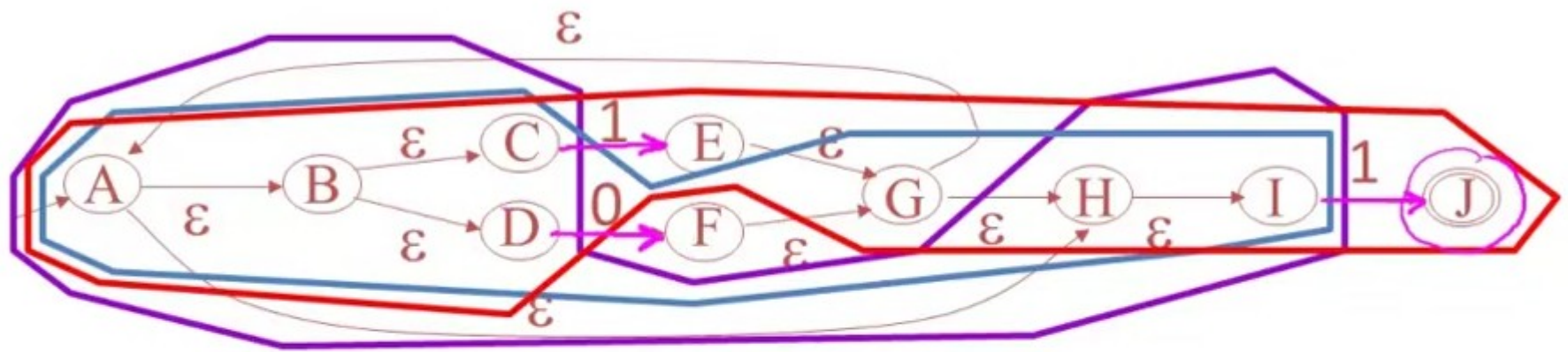




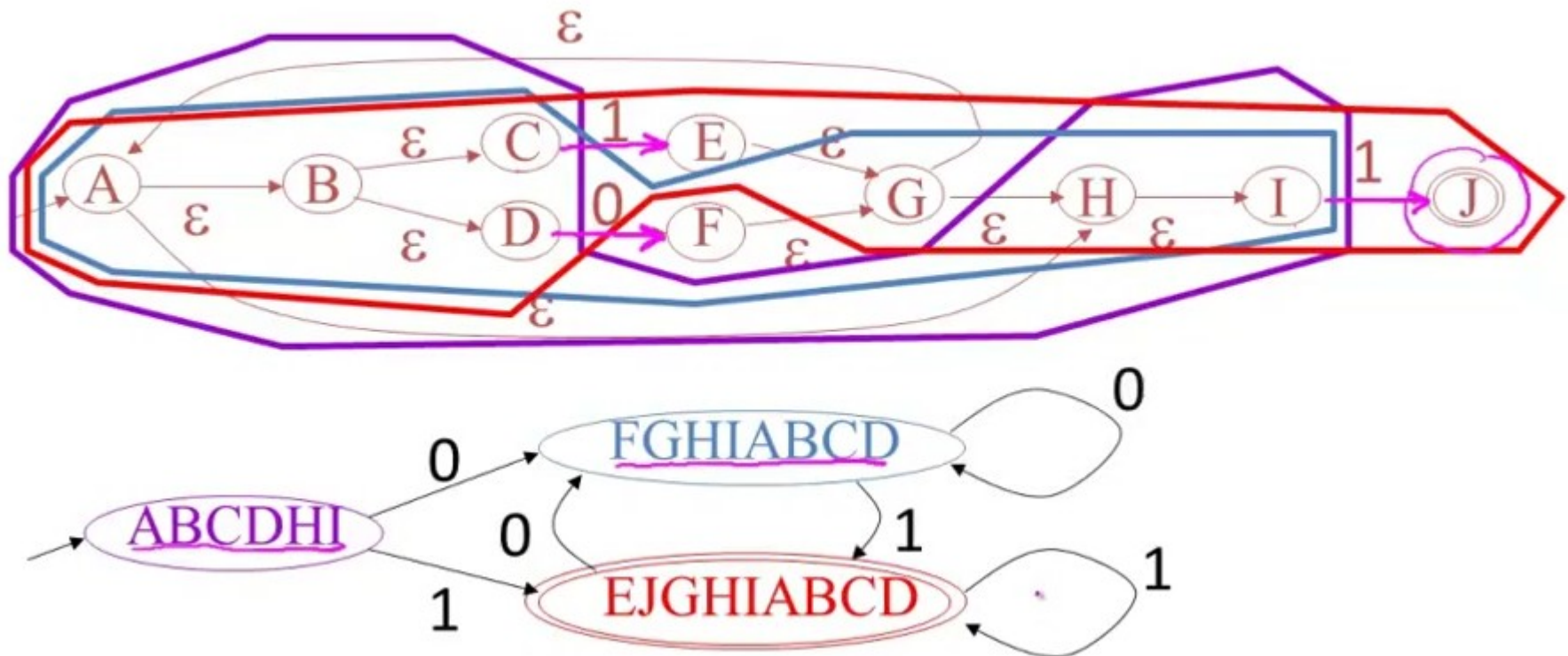
Epsilon closure of E and J







- Consider the regular expression

$$(1|0)^*1$$


- Regular languages
 - The weakest formal languages widely used
 - Many applications

Consider the language:

$$\{()^i \mid i \geq 0\}$$

()
 (())
 ((()))
 ⋮

Set of all balanced parentheses

((1+2) * 3)
 if then
 if then
 if then
 fi
 fi
 fi

What can regular languages express?



count mod k

$(^i)^i$

Regular languages
can check parity

Cannot count arbitrarily high