Programming Languages & Translators

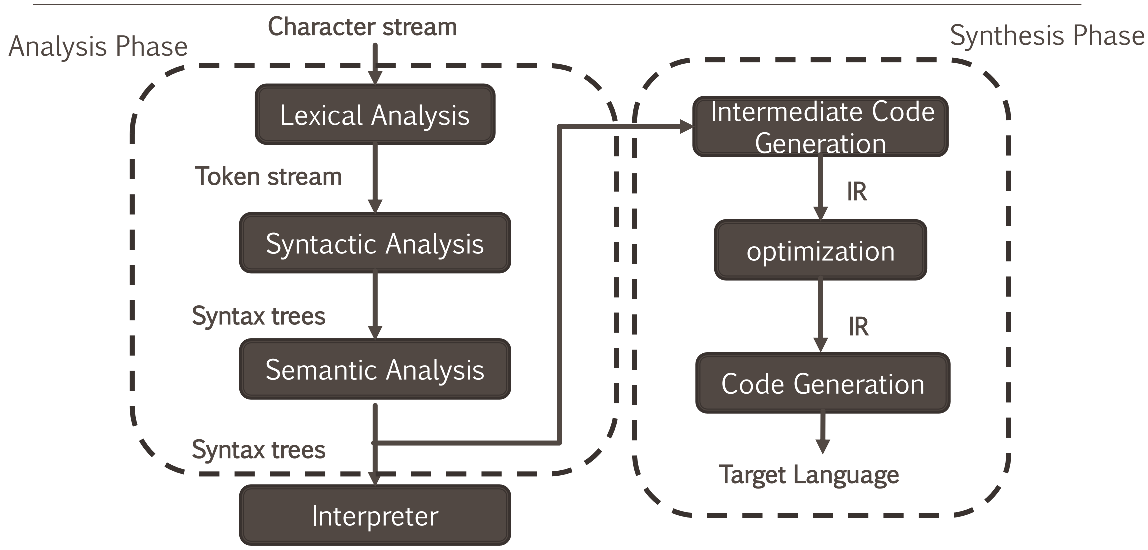
LEXICAL ANALYSIS

Baishakhi Ray

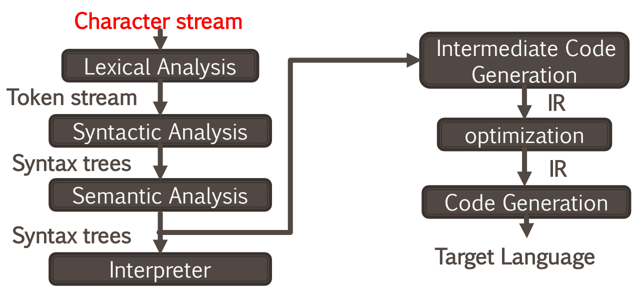
Fall 2019

These slides are motivated from Prof. Alex Aiken: Compilers (Stanford)

Structure of a Typical Compiler



Input to Compiler

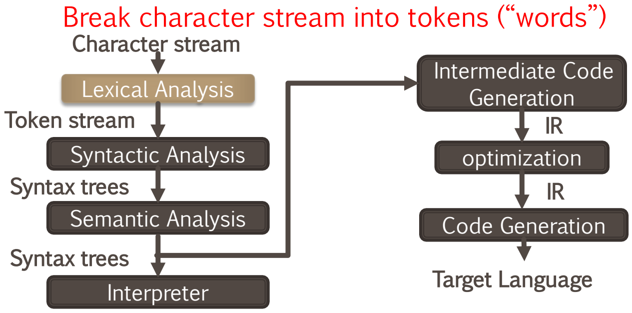



```
if(i == j)
  z = 0;
else
  z = 1;
```

i f (i = = j)\n\t z = 0 ; \n\t else   z = 1 ;

Lexical Analysis

Break character stream into tokens (“words”)




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

i f (i = = j)\n\t z = 0 ; \n\t else   z = 1 ;

1. Identify the substrings

‘if’ ‘(’ ‘i’ ‘==’ ‘j’ ‘)’ ‘\n’ ‘\t’ ‘z’ ‘=’ ‘0’ ‘;’ ‘\n’ ‘\t’ ‘else’ ‘z’ ‘=’ ‘1’ ‘;’

2. Identify the token classes

keyword<if> LPAR identifier<i> op<==> identifier<j> RPAR
whitespaces identifier<z> op<=> number<0> <;> whitespaces
keyword<else> identifier<z> op<=> number<1> ‘;’

Token Class

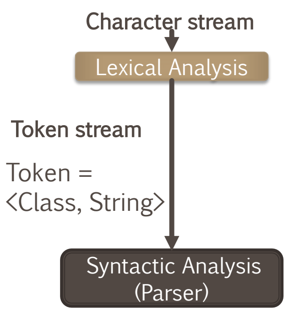
- In Programming Language
  - keywords, identifiers, LPAR, RPAR, const, etc.
- In English?
  - Noun, verb, …

Token Class

- Each class corresponds to a set of strings
- Identifier
  - Strings are letters or digits, starting with a letter
  - Eg:
- Numbers:
  - A non-empty strings of digits
  - Eg:
- Keywords
  - A fixed set of reserved words
  - Eg:
- Whitespace
  - A non-empty sequence of blanks, newlines, and tabs

Lexical Analysis (Example)

- Classify program substrings according to roles (token class)
- Communicate tokens to parser



Z = 1

- <Id, “Z”>
- <Op, “=”>
- <Numbers, “1”>

“Z”, “=”, “1” are called lexemes (an instance of the corr. token class)

Lexical Analysis: Examples

Here is a photo of my house;
<p>
see More Picture if you liked that one.</p>

<text, "Here is a photo of">
<nodestart, b>
<text, "my house">
<nodeend, b>
<nodestart, p>
<selfendnode, img>
<selfendnode, br>
<text, "see">
<nodestart, a>
<text, "More Picture">
<nodeend, a>
<text, "if you liked that one.">
<nodeend, p>

Lexical Analysis: Examples

- Usually, given the **pattern** describing the lexemes of a token, it is relatively simple to recognize matching lexemes when they occur on the input.
- However, in some languages, it is not immediately apparent when we have seen an instance of a lexeme corresponding to a token.

```
FORTRAN RULE: White Space is insignificant: VA R1 == VAR1

DO 5 I = 1.25

DO 5 I = 1,25
```

- Lexical analysis may require to "look ahead" to resolve ambiguity.
  - Look ahead complicates the design of lexical analysis
  - Minimize the amount of look ahead

Lookahead

- Lexical analysis tries to partition the input string into the logically units of the language. This is implemented by reading left to right. "scanning", recognizing one token at a time.
- "Lookahead" is required to decide where one token ends and the next token begins.

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

==/=?

Keyword/Identifier?

Lexical Analysis: Examples

- C++ template Syntax:
  - Foo<Bar>
- C++ stream Syntax:
  - cin >> var
- Ambiguity
  - Foo<Bar<Bar1>>
  - cin >> var

Summary So Far

- The goal of Lexical Analysis
  - Partition the input string to lexeme
  - Identify the token class of each lexeme
- Left-to-right scan => look ahead may require

REGULAR LANGUAGES

- Lexical structure of a programming language is a set of token classes.
- Each token class consists of some set of strings.
- How to map which set of strings belongs to which token class?
  - Use regular languages
- Use Regular Expressions to define Regular Languages.

Regular Expressions

- Single character
  - 'c' = {"c"}
- Epsilon
  - ε = {""}
- Union
  - A + B = {a | a ϵ A} ∪ {b | b ϵ B}
- Concatenation
  - AB = {ab | a ϵ A ^ b ϵ B}
- Iteration (Kleene closure)
  - A* = $\cup_{i\geq 0} A^i$ = A.....A (i times)
  - $A^0 = \varepsilon$ (empty string)

Regular Expressions

- **Def:** The regular expressions over Σ are the smallest set of expressions including

R = ε

| 'c', 'c' ϵ Σ

| R + R

| RR

| R^*

Regular Expression Example

- Σ = {0,1}
  - 1*
  - (0+1)1
  - 0*+1*
  - (0+1)*

- There can be many ways to write an expression

Formal Languages

- **Def**: Let Σ be a set of character (alphabet}. A language over Σ is a set of strings of characters drawn from Σ.
  - Regular languages is a formal language
- Alphabet = English character, Language = English Language
  - Is it formal language?
- Alphabet = ASCII, Language = C Language

Formal Language

‘c’ = {“c”}
ε = {“”}
A + B = {a | a ϵ A} ∪ {b | b ϵ B}
AB = {ab | a ϵ A ^ b ϵ B}
$\underbrace{A^*}_{\text{expression}} = \underbrace{\bigcup_{i\geq 0} A^i}_{\text{Set}}$

Formal Language

L(‘c’) = {“c”}
L(ε) = {“”}
L(A + B) = {a | a ϵ L(A)} ∪ {b | b ϵ L(B)}
L(AB) = {ab | a ϵ L(A) ^ b ϵ L(B)}
$\underbrace{L(A^*)}_{\text{expression}} = \underbrace{\bigcup_{i\geq 0} L(A^i)}_{\text{Set}}$

L: Exp -> Set of strings

- Meaning function L maps syntax to semantics
- Mapping is many to one
- Never one to many

Lexical Specifications

- Keywords: “if” or “else” or “then” or “for” ….
  - Regular expression = ‘i’ ‘f’ + ‘e’ ‘l’ ‘s’ ‘e’
    = ‘if’ + ‘else’ + ‘then’

- Numbers: a non-empty string of digits
  - digit = ‘0’+’1’+’2’+’3’+’4’+’5’+’6’+’7’+’8’+’9’
  - digit*
  - How to enforce *non-empty string*?
    - digit digit* = digit$^+$

Lexical Specifications

- Identifier: strings of letters or digits, starting with a letter
  - letter = ‘a’ + ‘b’ + ‘c’ + …. + ‘z’ + ‘A’ + ‘B’ + …. + ‘Z’
    = [a-zA-Z]
  - letter (letter + digit)*

- Whitespace: a non-empty sequence of blanks, newline, and tabs
  - (‘ ’ + ‘\n’ + ‘\t’)$^+$

PASCAL Lexical Specification

- digit = ‘0’+’1’+’2’+’3’+’4’+’5’+’6’+’7’+’8’+’9’
- digits = digit$^+$
- opt_fraction = (‘.’ digits) + ε = (‘.’ digits)?
- opt_exponent = (‘E’ (‘+’ + ‘-’ + ε) digits) + ε
  = (‘E’ (‘+’ + ‘-’)? digits)?
- num = digits opt_fraction opt_exponent

Common Regular Expression

- At least one $A^+ \equiv AA^*$
- Union: A | B ≡ A + B
- Option: A? ≡ A + ε
- Range: ‘a’ + … + ‘z’ = [a-z]
- Excluded range: complement of [a-z] ≡ [^a-z]

Lexical Specification of a language

1. Write a regex for the lexemes of each token class
   - Number = $digit^+$
   - Keywords = ‘if’ + ‘else’ + ..
   - Identifiers = letter (letter + digit)*
   - LPAR = ‘(‘

Lexical Specification of a language

2. Construct **R**, matching all lexemes for all tokens

   R = Number + Keywords + Identifiers + …

   $= R_1 + R_2 + R_3 + \ldots$

3. Let input be $x_1 \ldots x_n$.

   For $1 \leq i \leq n$, check $x_1 \ldots x_i \in L(R)$

4. If successful, 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 step 3.

Lexical Specification of a language

- How much input is used?
  - $x_1 \ldots x_i \in L(R)$
  - $x_1 \ldots x_j \in L(R)$, $i \neq j$
  - Which one do we want? (e.g., == or =)
  - Maximal munch: always choose the longer one

- Which token is used if more than one matches?
  - $x_1 \ldots x_i \in L(R)$ where $R = R_1 + R_2 + .. + R_n$
  - $x_1 \ldots x_i \in L(R_m)$
  - $x_1 \ldots x_i \in L(R_n)$, $m \neq n$
  - Eg: Keywords = ‘if’, Identifier = letter (letter + digit)*, if matches both
  - Keyword has higher priority
  - Rule of Thumb: Choose the one listed first

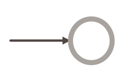
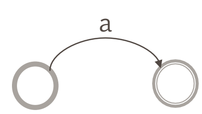
Lexical Specification of a language

- What if no rule matches?
  - $x_1 \ldots x_i \notin L(R)$ … compiler typically tries to avoid this scenario
  - Error = [all strings not in the lexical spec]
  - Put it in last in priority

Summary so far

- Regular Expressions are concise notations for the string patterns
- Use in lexical analysis with some extensions
  - To resolve ambiguities
  - To handle errors
- Implementation?
  - We will study next

Finite Automata

- Regular Expression = specification
- Finite Automata = implementation
- 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 state: state1 $\xrightarrow{input}$ state2 

Transition

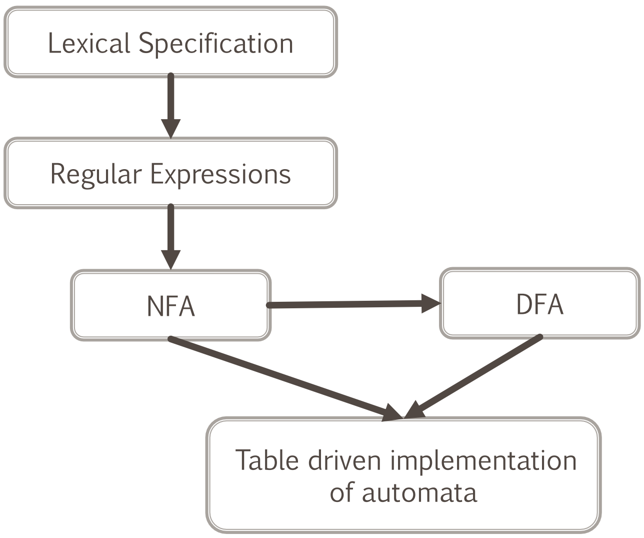
- s1 $\xrightarrow{a}$ s2 (state s1 on input a goes to state s2)
- If end of the input and in final state, the input is accepted
- Otherwise reject

- Language of FA = set of strings accepted by that FA

Example Automata

- a finite automaton that accepts only "1"

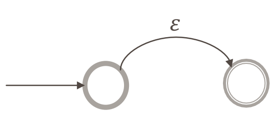
- A finite automaton that accepting any number of "1" followed by "0"



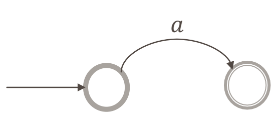
Regular Expression to NFA

- For each kind of regex, define a NFA
  - Notation: NFA for regex M

- For ε

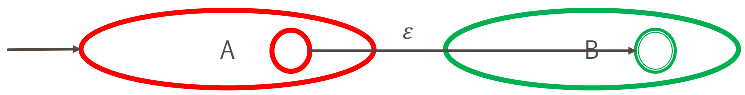


- For input a

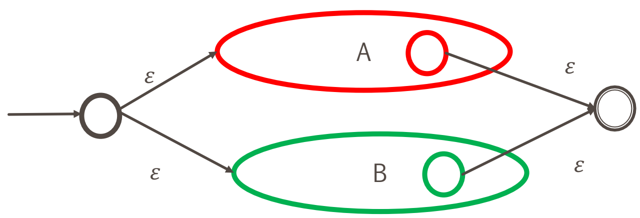


Regular Expression to NFA

- For AB

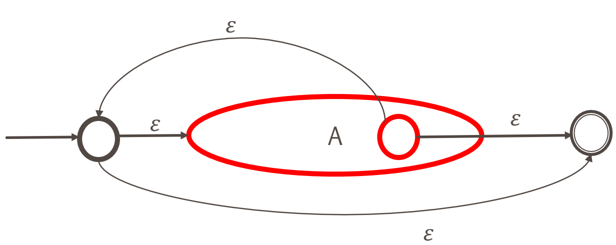


- For A + B



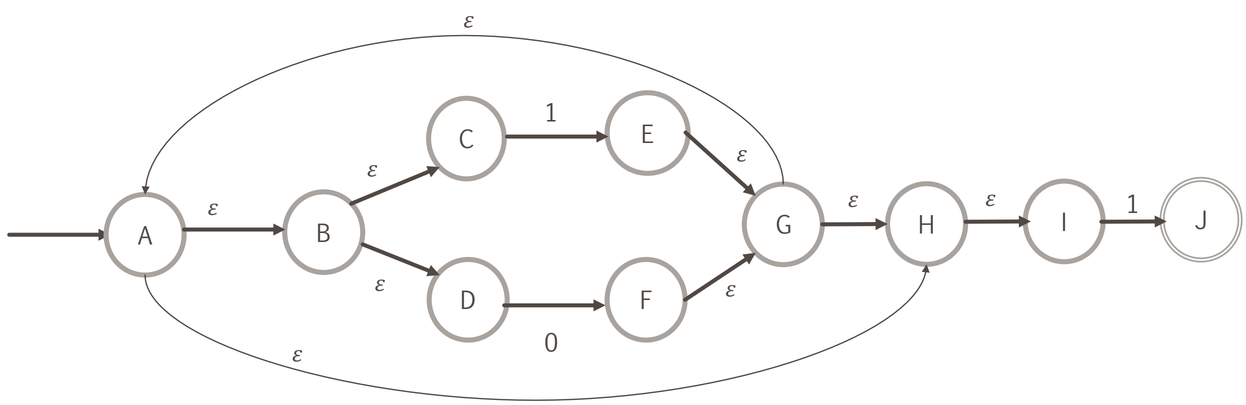
Regular Expression to NFA

- For A*

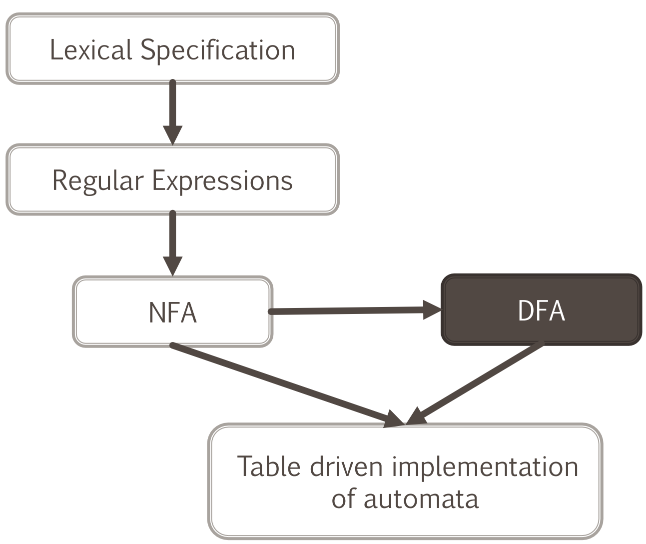


Example

- (1+0)*1

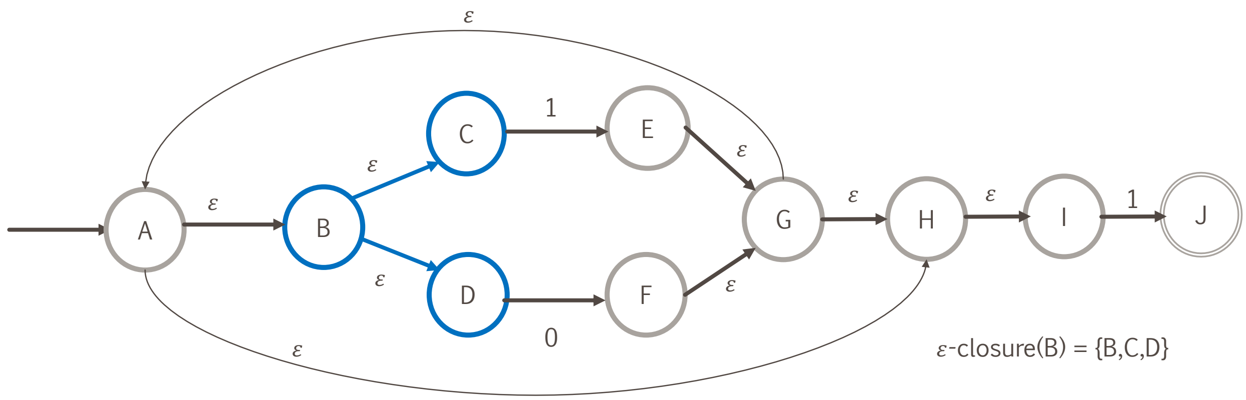


NFA to DFA



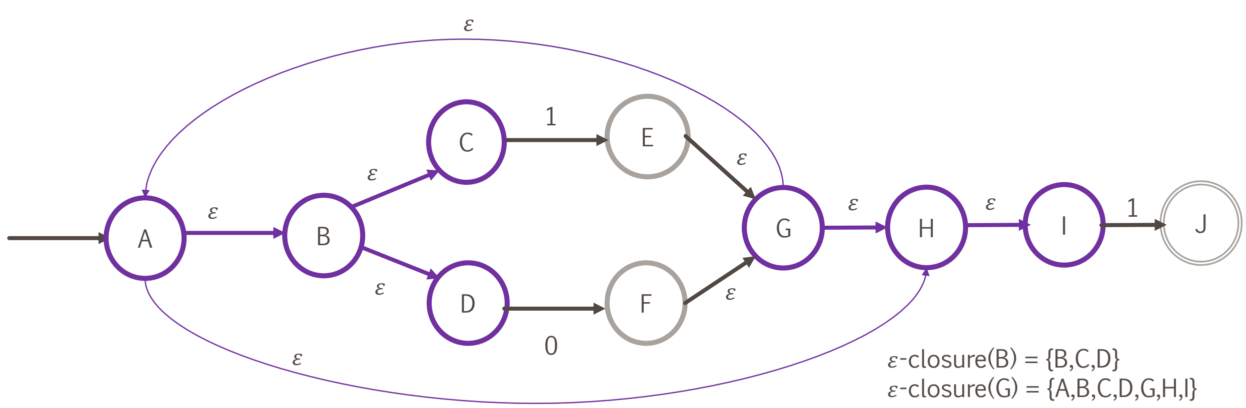
ε-closure

- ε-closure of a state is all the state I can reach following ε move.



ε-closure

- ε-closure of a state is all the state I can reach following ε move.



NFA

- An NFA can be in many states at any time

- How many different states?
  - If NFA has N states, it reaches some subset of those states, say S
  - $|S| \leq N$
  - There are $2^N - 1$ possible subsets (finite number)

NFA to DFA

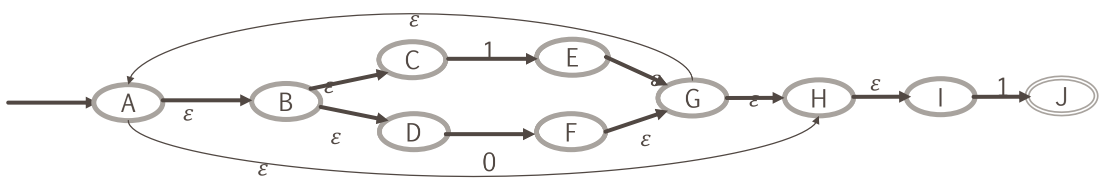
NFA

- States S
- Start s
- Final state F
- Transition state
  - $a(X) = \{y \mid x \in X \wedge x \xrightarrow{a} y)$
- $\varepsilon - closure$

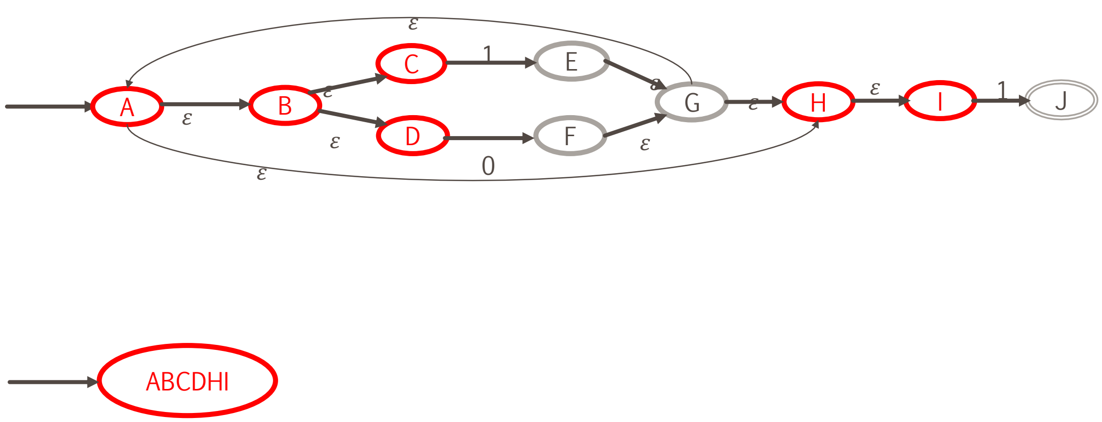
DFA

- States will be all possible subset of S except empty set
- Start state = $\varepsilon - closure(s)$
- Final state $\{X \mid X \cap F = \emptyset\}$
- $X \xrightarrow{a} Y$ if
  - $Y = \varepsilon - closure\ (a(X))$

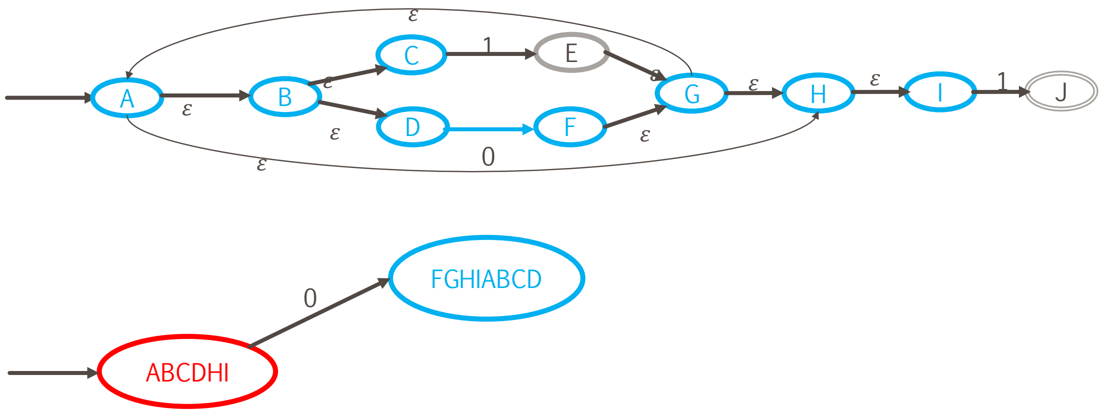
NFA to DFA



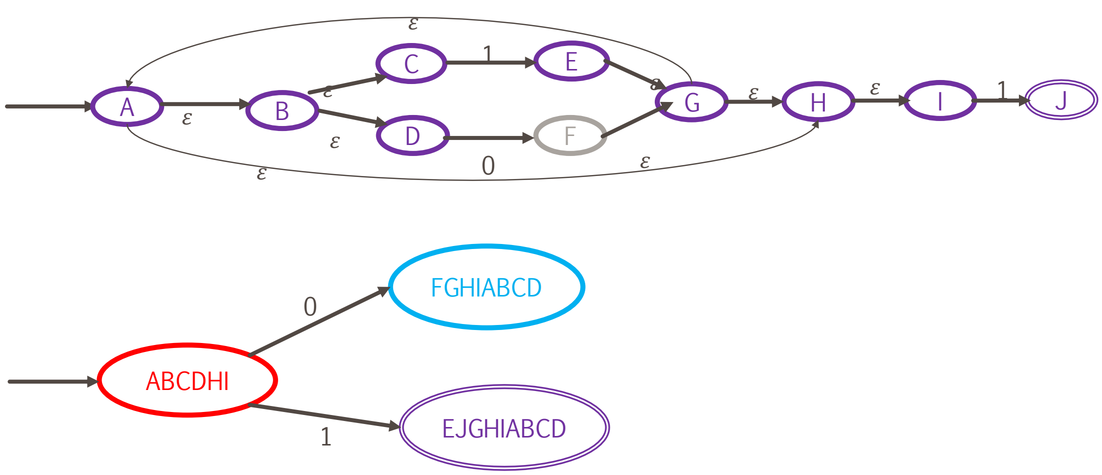
NFA to DFA



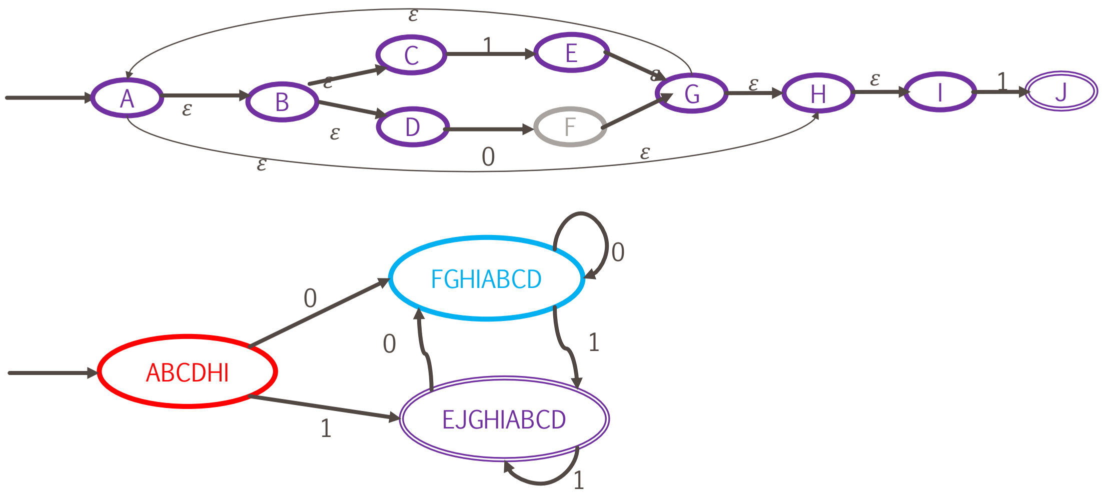
NFA to DFA



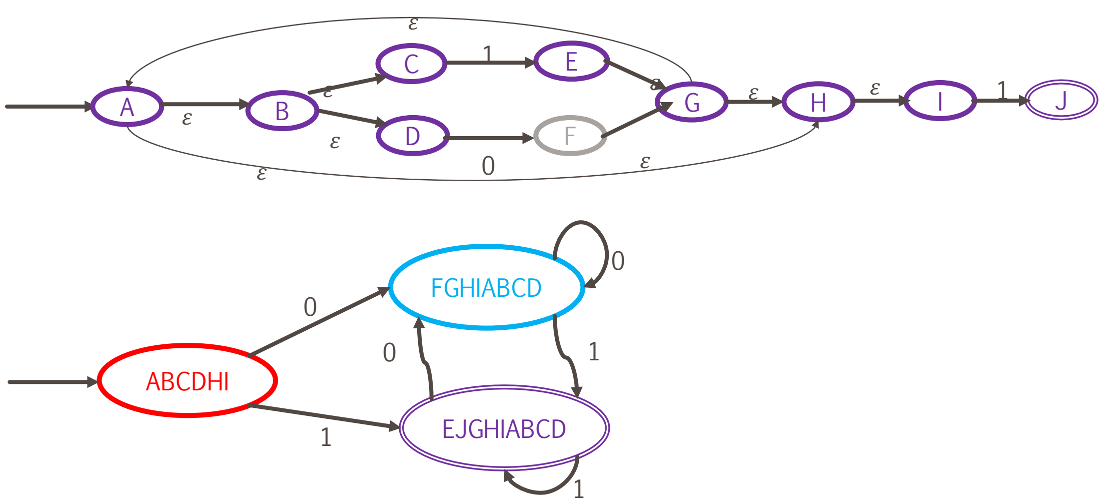
NFA to DFA



NFA to DFA



NFA to DFA



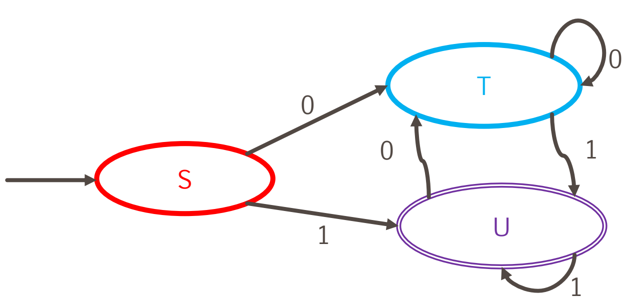
Implementing DFA

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

Implementing DFA

Table A

	0	1
S	T	U
T	T	U
U	T	U




```
i = 0;
state = 0;
while(input[i])
  state = A[state,input[i]]
```

Implementing DFA

Table A

	0	1
S	T	U
T	T	U
U	T	U

A lot of duplicate entries

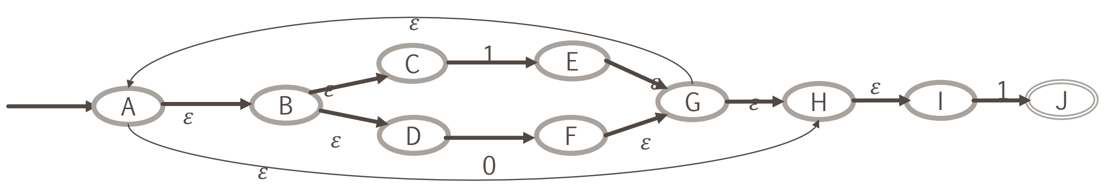
Table B

S
T
U

0	1
T	U

Compact but need an extra indirection
- Inner loop will be slower

Implementing DFA



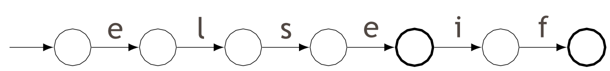
	0	1	ε
A			{B,H}
B			{C,D}
C		{E}	
…			

Deal with set of states rather than single state-→ inner loop is complicated

Deterministic Finite Automata: Example

```
{
   type token = ELSE | ELSEIF
}

rule token =
  parse "else"{ ELSE }
      | "elseif"{ ELSEIF }
```



Deterministic Finite Automata

```
{ type token = IF | ID of string | NUM of string }

rule token =
  parse "if"{ IF }
      | ['a'-'z'] ['a'-'z' '0'-'9']* as lit { ID(lit) }
      | ['0'-'9']+                    as num { NUM(num) }
```