Scanner

Ronghui Gu

Spring 2024

Columbia University

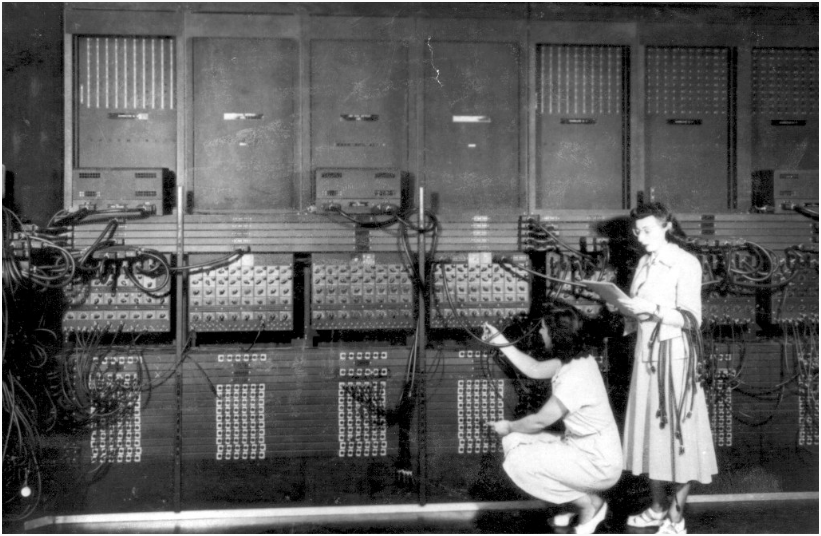
* Course website: https://verigu.github.io/4115Spring2024/

The Big Picture

The First Question

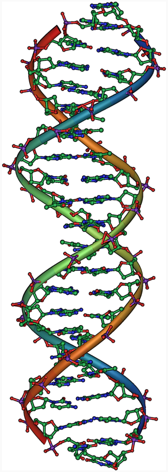
How do we describe/construct a program?

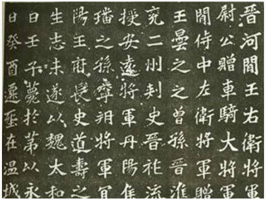
The ENIAC: Programming with Spaghetti

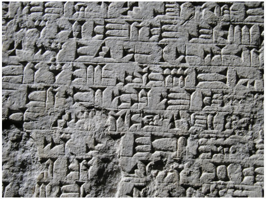


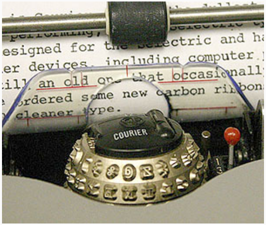
Solution: Use a Discrete Combinatorial System

Use combinations of a small number of things to represent (exponentially) many different things.





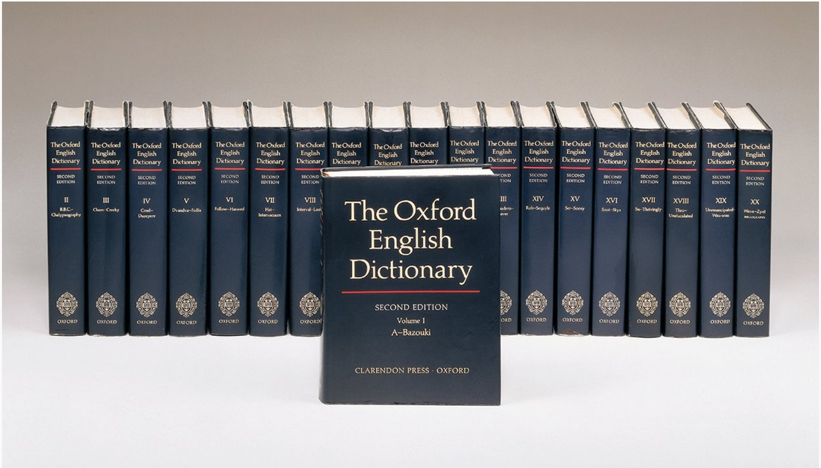




The Second Question

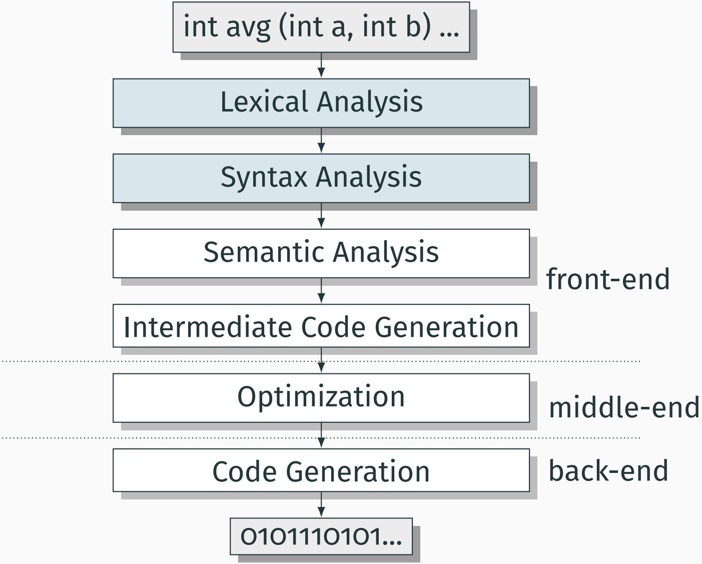
How do we describe the combinations of a small number of things.

Just List Them?



Gets annoying for large numbers of combinations

Scanning and Parsing



Lexical Analysis

Lexical Analysis (Scanning)

Translate a stream of characters to a stream of tokens

$\mathrm{f\ o\ o} \llcorner = \llcorner \mathrm{a} + \llcorner \mathrm{bar} (0 , \llcorner 42 , \llcorner \mathrm{q}) ;$

$\boxed{\text{ID}}$ $\boxed{\text{EQUALS}}$ $\boxed{\text{ID}}$ $\boxed{\text{PLUS}}$ $\boxed{\text{ID}}$ $\boxed{\text{LPAREN}}$ $\boxed{\text{NUM}}$ $\boxed{\text{COMMA}}$ $\boxed{\text{ID}}$ $\boxed{\text{LPAREN}}$ $\boxed{\text{SEMI}}$

Token	**Lexemes**	**Pattern**
EQUALS	$=$	an equals sign
PLUS	$+$	a plus sign
ID	a foo bar	letter followed by letters or digits
NUM	0 42	one or more digits

Lexical Analysis

Goal: simplify the job of the parser and reject some wrong programs, e.g.,

```
%#$^#!#%#$
```

is not a C program†

Lexical Analysis

Goal: simplify the job of the parser and reject some wrong programs, e.g.,

```
%#$^#!#%#$
```

is not a C program†

Scanners are usually much faster than parsers.

Lexical Analysis

Goal: simplify the job of the parser and reject some wrong programs, e.g.,

```
%#$^#!#%#$
```

is not a C program†

Scanners are usually much faster than parsers.

Discard as many irrelevant details as possible (e.g., whitespace, comments).

Lexical Analysis

Goal: simplify the job of the parser and reject some wrong programs, e.g.,

```
%#$^#!#%#$
```

is not a C program[†]

Scanners are usually much faster than parsers.

Discard as many irrelevant details as possible (e.g., whitespace, comments).

Parser does not care that the identifer is "supercalifragilisticexpialidocious."

Parser rules are only concerned with tokens.

[†] It is what you type when your head hits the keyboard

Describing Tokens

Alphabet: A finite set of symbols

Examples: $\{$ 0, 1 $\}$, $\{$ A, B, C, $\ldots$, Z $\}$, ASCII, Unicode

Describing Tokens

Alphabet: A finite set of symbols

Examples: { 0, 1 }, { A, B, C, ..., Z }, ASCII, Unicode

String: A finite sequence of symbols from an alphabet

Examples: ϵ (the empty string), Ronghui, $\alpha\beta\gamma$

Describing Tokens

Alphabet: A finite set of symbols

Examples: $\{$ 0, 1 $\}$, $\{$ A, B, C, . . . , Z $\}$, ASCII, Unicode

String: A finite sequence of symbols from an alphabet

Examples: ϵ (the empty string), Ronghui, $\alpha\beta\gamma$

Set: A set of strings over an alphabet

Examples: $\emptyset$ (the empty language), $\{$ 1, 11, 111, 1111 $\}$, all English words, strings that start with a letter followed by any sequence of letters and digits

Operations on String Sets

Let $L = \{ \epsilon, \text{wo} \}$, $M = \{ \text{man}, \text{men} \}$

Concatenation: Strings from one followed by the other

$LM = \{ \text{man}, \text{men}, \text{woman}, \text{women} \}$

Operations on String Sets

Let $L = \{ \epsilon, \text{wo} \}$, $M = \{ \text{man}, \text{men} \}$

Concatenation: Strings from one followed by the other

$LM = \{ \text{man}, \text{men}, \text{woman}, \text{women} \}$

Union: All strings from each string set

$L \cup M = \{\epsilon, \text{wo}, \text{man}, \text{men} \}$

Operations on String Sets

Let $L = \{ \epsilon$, wo $\}$, $M = \{$ man, men $\}$

Concatenation: Strings from one followed by the other

$LM = \{$ man, men, woman, women $\}$

Union: All strings from each string set

$L \cup M = \{\epsilon,$ wo, man, men $\}$

Kleene Closure: Zero or more concatenations

$M^* = \{\epsilon\} \cup M \cup MM \cup MMM \cdots =$
$\{\epsilon,$ man, men, manman, manmen, menman, menmen, manmanman, manmanmen, manmenman, $\ldots\}$

Regular Expressions over an Alphabet Σ

A standard way to express tokens.

1. ϵ is a regular expression that denotes $\{\epsilon\}$
2. If $a \in \Sigma$, a is an RE that denotes $\{a\}$
3. If r and s denote sets $L(r)$ and $L(s)$,

Regular Expressions over an Alphabet Σ

A standard way to express tokens.

1. ϵ is a regular expression that denotes $\{\epsilon\}$
2. If $a \in \Sigma$, a is an RE that denotes $\{a\}$
3. If r and s denote sets $L(r)$ and $L(s)$,

$$(r) \mid (s) \quad \text{denotes} \quad L(r) \cup L(s)$$

Regular Expressions over an Alphabet Σ

A standard way to express tokens.

1. ϵ is a regular expression that denotes $\{\epsilon\}$
2. If $a \in \Sigma$, a is an RE that denotes $\{a\}$
3. If r and s denote sets $L(r)$ and $L(s)$,

$$\begin{array}{lll} (r) \mid (s) & \text{denotes} & L(r) \cup L(s) \\ (r)(s) & & \{tu : t \in L(r), u \in L(s)\} \end{array}$$

Regular Expressions over an Alphabet Σ

A standard way to express tokens.

1. ϵ is a regular expression that denotes $\{\epsilon\}$
2. If $a \in \Sigma$, a is an RE that denotes $\{a\}$
3. If r and s denote sets $L(r)$ and $L(s)$,

$(r) \mid (s)$	denotes	$L(r) \cup L(s)$
$(r)(s)$		$\{tu : t \in L(r), u \in L(s)\}$
$(r)^*$		$\cup_{i=0}^{\infty} L(r)^i$
	where	$L(r)^0 = \{\epsilon\}$
	and	$L(r)^i = L(r)L(r)^{i-1}$

Regular Expression Examples

$\Sigma = \{a, b\}$

Regexp.	String Set
$a \mid b$	$\{a, b\}$
$(a \mid b)(a \mid b)$	

Regular Expression Examples

$\Sigma = \{a, b\}$

Regexp.	String Set
$a \mid b$	$\{a, b\}$
$(a \mid b)(a \mid b)$	$\{aa, ab, ba, bb\}$
$(a \mid b)^*$	

Regular Expression Examples

$\Sigma = \{a, b\}$

Regexp.	String Set
$a \mid b$	$\{a, b\}$
$(a \mid b)(a \mid b)$	$\{aa, ab, ba, bb\}$
$(a \mid b)^*$	$\{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, \ldots\}$
$a \mid a^*b$	$\{a, b, ab, aab, aaab, aaaab, \ldots\}$

Specifying Tokens with REs

ID: letter followed by letters or digits

Typical choice: $\Sigma =$ ASCII characters, i.e., $\{\text{␣}, !, ", \#, \$, \ldots, 0, 1, \ldots, 9, \ldots, A, \ldots, Z, \ldots, \tilde{}\}$

letters: $A \mid B \mid \cdots \mid Z \mid a \mid \cdots \mid z$

digits: $0 \mid 1 \mid \cdots \mid 9$

identifier:

Specifying Tokens with REs

ID: letter followed by letters or digits

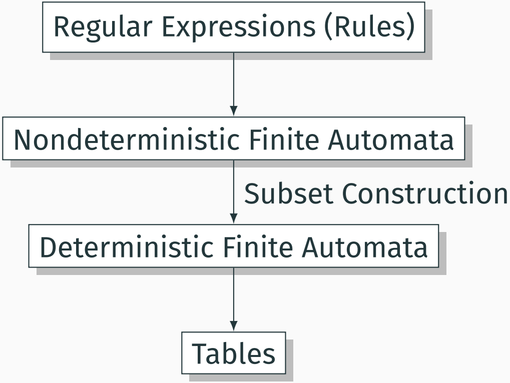
Typical choice: Σ = ASCII characters, i.e., $\{␣, !, ", \#, \$, \ldots, 0, 1, \ldots, 9, \ldots, A, \ldots, Z, \ldots, \tilde{}\}$

letters: $A \mid B \mid \cdots \mid Z \mid a \mid \cdots \mid z$

digits: $0 \mid 1 \mid \cdots \mid 9$

identifier: **letter** (**letter** | **digit**)*

Implementing Scanners Automatically

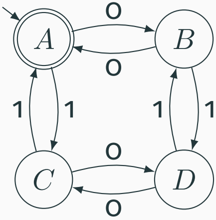


Finite Automata

“All strings containing an even number of 0’s and 1’s”

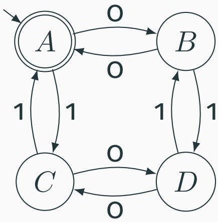
Finite Automata

"All strings containing an even number of 0's and 1's"



Finite Automata

"All strings containing an even number of 0's and 1's"



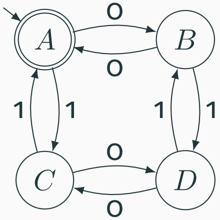
1. Set of states
   $S : \{ A \quad B \quad C \quad D \}$
2. Set of input symbols $\Sigma : \{0, 1\}$
3. Transition function
   $\sigma : S \times \Sigma_\epsilon \to 2^S$

state	ϵ	0	1
A	$\emptyset$	$\{B\}$	$\{C\}$
B	$\emptyset$	$\{A\}$	$\{D\}$
C	$\emptyset$	$\{D\}$	$\{A\}$
D	$\emptyset$	$\{C\}$	$\{B\}$

4. Start state s_0 : A
5. Set of accepting states
   $F : \{ A \}$

The String Set induced by an NFA

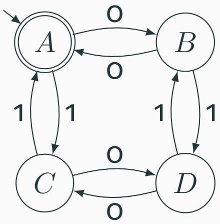
An NFA accepts an input string x iff there is a path from the start state to an accepting state that “spells out” x.



Show that the string 010010 is accepted.

The String Set induced by an NFA

An NFA accepts an input string x iff there is a path from the start state to an accepting state that “spells out” x.



Show that the string 010010 is accepted.

$$A \xrightarrow{0} B \xrightarrow{1} D \xrightarrow{0} C \xrightarrow{0} D \xrightarrow{1} B \xrightarrow{0} A$$

Translating REs into NFAs (Thompson's algorithm)

a

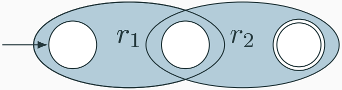
Symbol

Translating REs into NFAs (Thompson's algorithm)

a

Symbol

$r_1 r_2$



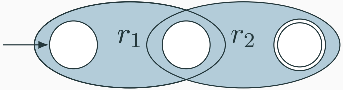
Sequence

Translating REs into NFAs (Thompson's algorithm)

a

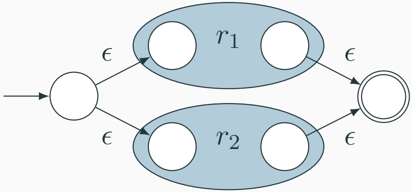
Symbol

$r_1 r_2$



Sequence

$r_1 \mid r_2$



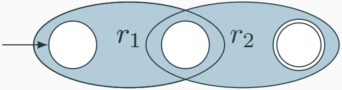
Choice

Translating REs into NFAs (Thompson’s algorithm)

a

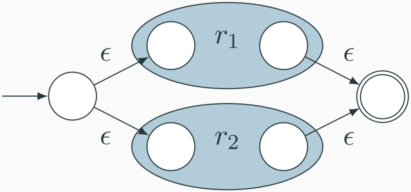
Symbol

$r_1 r_2$



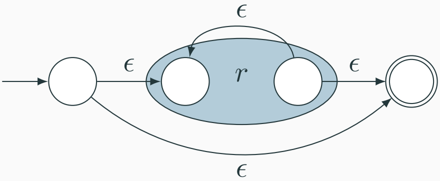
Sequence

$r_1 \mid r_2$



Choice

$(r)^*$

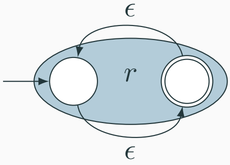


Kleene Closure

Why So Many Extra States and Transitions?

Invariant: Single start state; single end state; at most two outgoing arcs from any state: helpful for simulation.

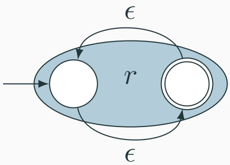
What if we used this simpler rule for Kleene Closure?



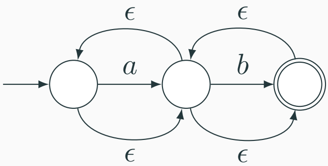
Why So Many Extra States and Transitions?

Invariant: Single start state; single end state; at most two outgoing arcs from any state: helpful for simulation.

What if we used this simpler rule for Kleene Closure?



Now consider a^*b^* with this rule:



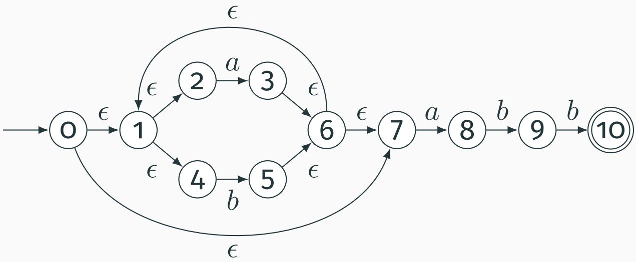
Is this right?

Translating REs into NFAs

Example: Translate $(a \mid b)^* abb$ into an NFA. Answer:

Translating REs into NFAs

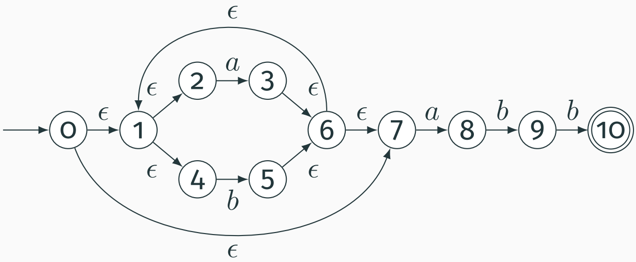
Example: Translate $(a \mid b)^*abb$ into an NFA. Answer:



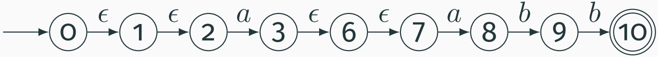
Show that the string "$aabb$" is accepted. Answer:

Translating REs into NFAs

Example: Translate $(a \mid b)^* abb$ into an NFA. Answer:



Show that the string "$aabb$" is accepted. Answer:



Simulating NFAs

Problem: you must follow the "right" arcs to show that a string is accepted. How do you know which arc is right?

Simulating NFAs

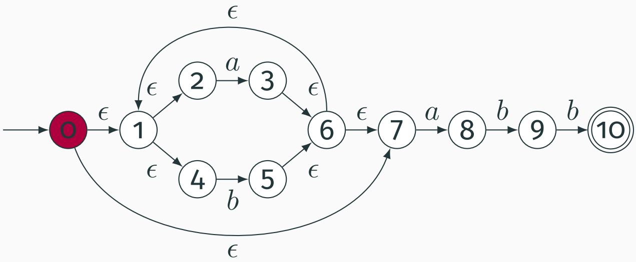
Problem: you must follow the "right" arcs to show that a string is accepted. How do you know which arc is right?

Solution: follow them all and sort it out later.

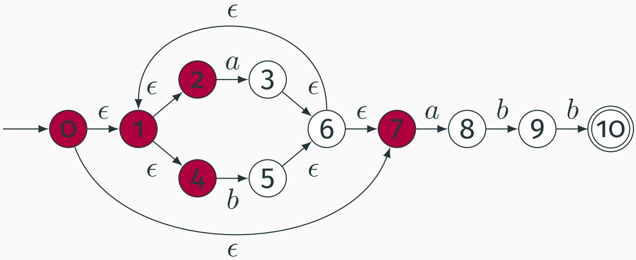
"Two-stack" NFA simulation algorithm:

1. Initial states: the ϵ-closure of the start state
2. For each character c,
   - New states: follow all transitions labeled c
   - Form the ϵ-closure of the current states
3. Accept if any final state is accepting

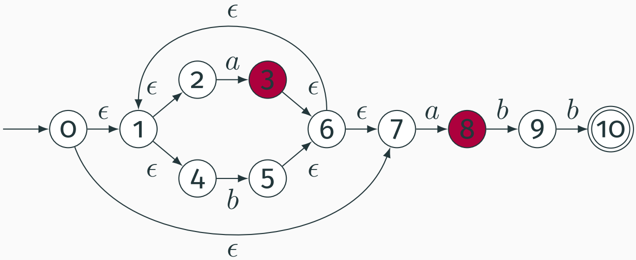
Simulating an NFA: $\cdot aabb$, Start



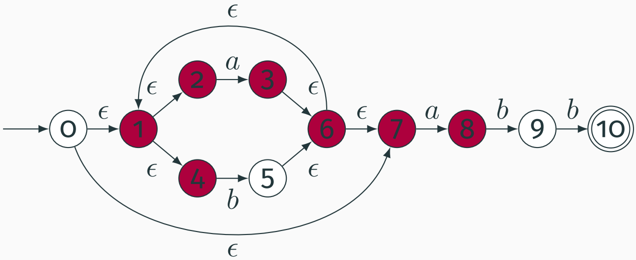
Simulating an NFA: $\cdot aabb$, ϵ-closure



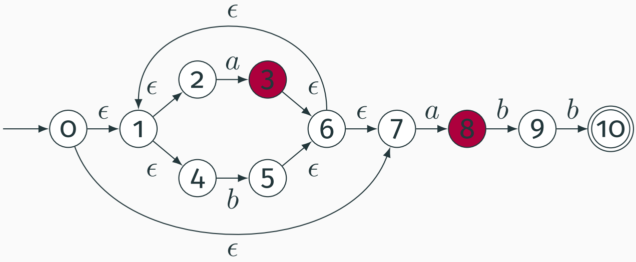
Simulating an NFA: $a{\cdot}abb$



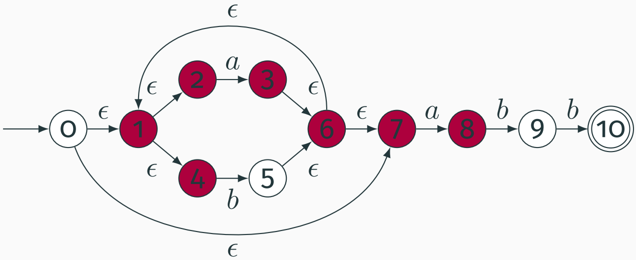
Simulating an NFA: $a{\cdot}abb$, ϵ-closure



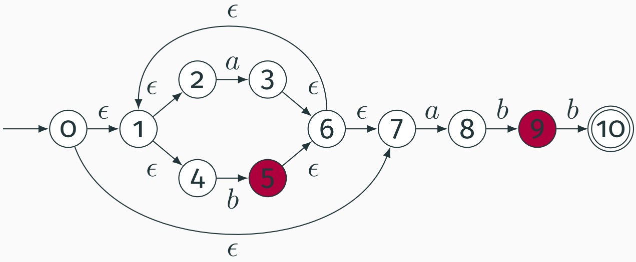
Simulating an NFA: $aa \cdot bb$



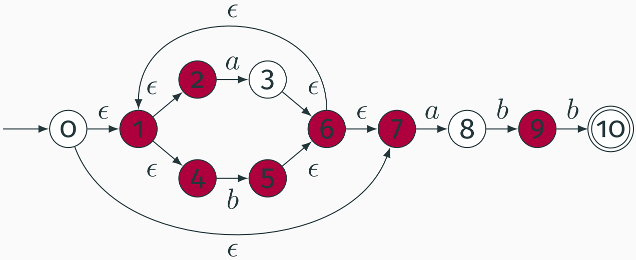
Simulating an NFA: $aa{\cdot}bb$, ϵ-closure



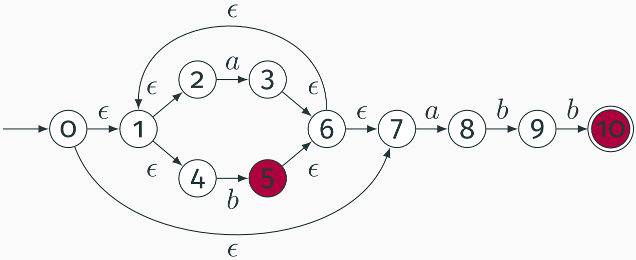
Simulating an NFA: $aab{\cdot}b$



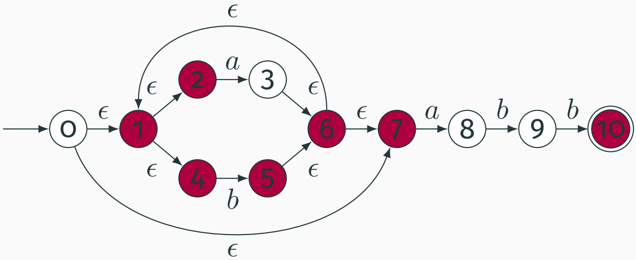
Simulating an NFA: $aab{\cdot}b$, ϵ-closure



Simulating an NFA: $aabb\cdot$



Simulating an NFA: $aabb\cdot$, Done



Deterministic Finite Automata

Restricted form of NFAs:

- No state has a transition on ϵ
- For each state s and symbol a, there is at most one edge labeled a leaving s.

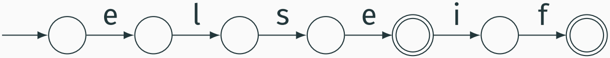
Very easy to check acceptance: simulate by maintaining current state. Accept if you end up on an accepting state. Reject if you end on a non-accepting state or if there is no transition from the current state for the next symbol.

Deterministic Finite Automata

```
{
   type token = ELSE | ELSEIF
}

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

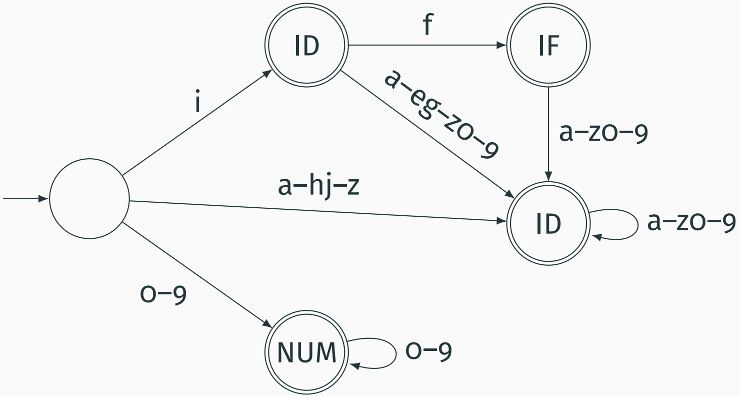
Deterministic Finite Automata

```
{
   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) }
```



Building a DFA from an NFA

Subset construction algorithm

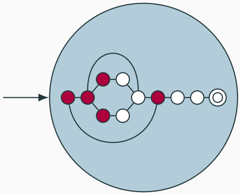
Simulate the NFA for all possible inputs and track the states that appear.

Each unique state during simulation becomes a state in the DFA.

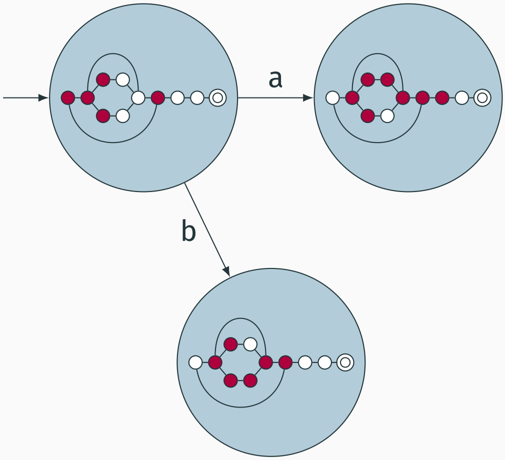
The Subset Construction Algorithm

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:
   - Apply move to the newly-created state and the input symbol; this will return a set of states.
   - 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.

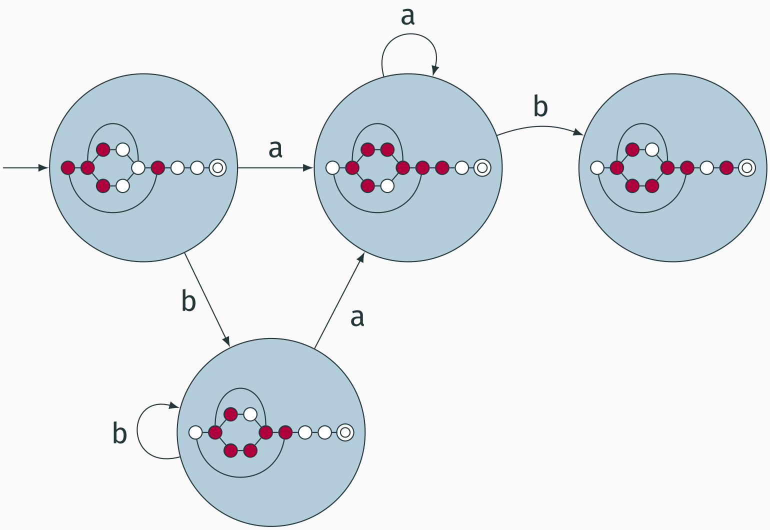
Subset construction for $(a \mid b)^*abb$



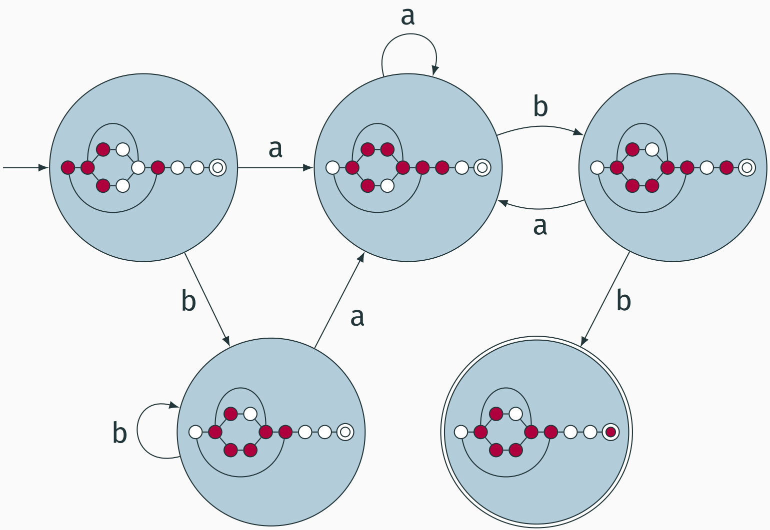
Subset construction for $(a \mid b)^*abb$



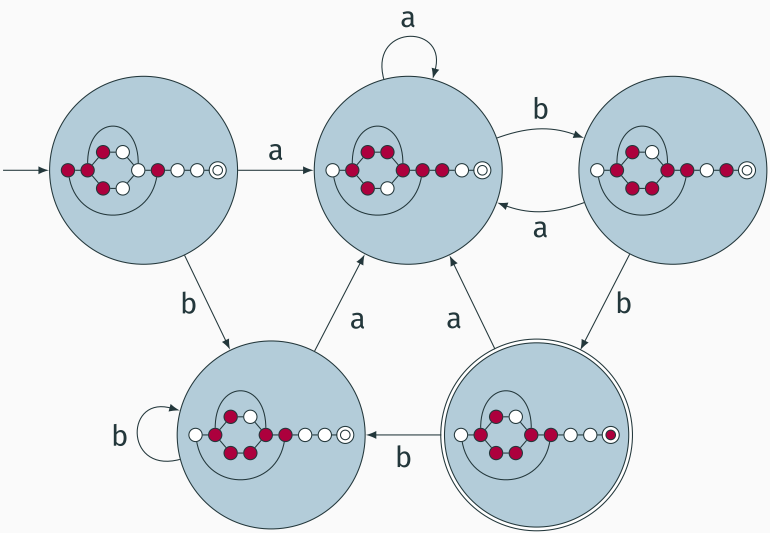
Subset construction for $(a \mid b)^*abb$



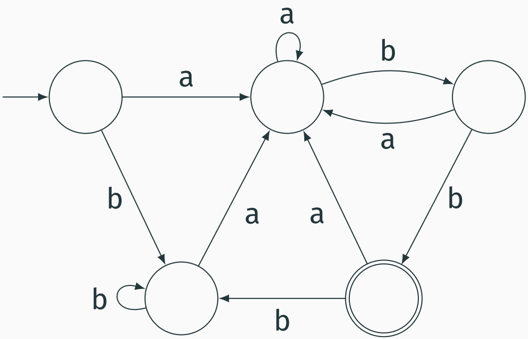
Subset construction for $(a \mid b)^* abb$



Subset construction for $(a \mid b)^*abb$

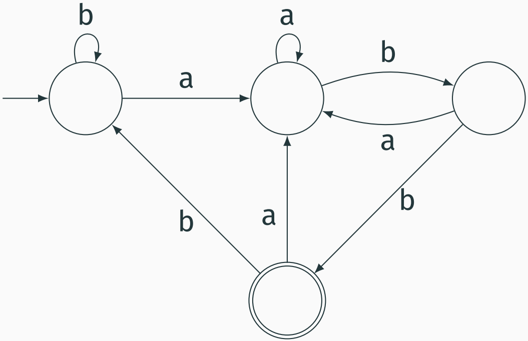


Result of subset construction for $(a \mid b)^*abb$



Is this minimal?

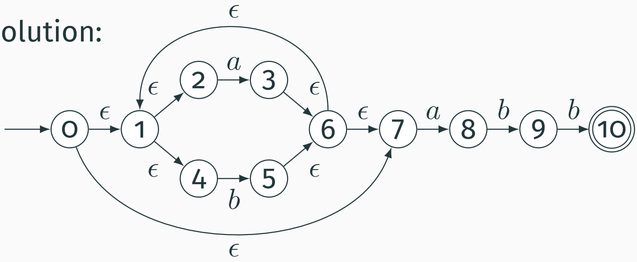
Minimized result for $(a \mid b)^*abb$



Transition Table Used In the Dragon Book

Problem: Translate $(a \mid b)^{*}abb$ into a DFA.

Solution:

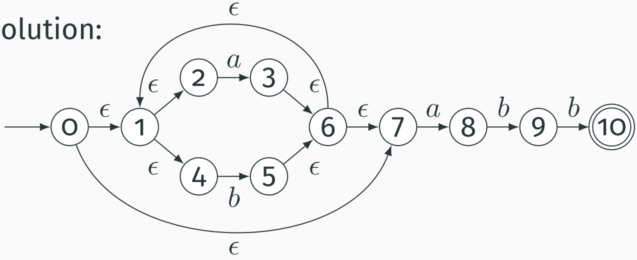


NFA State	DFA State	a	b

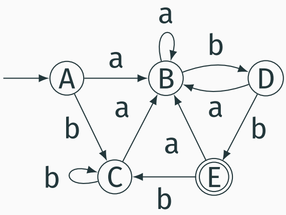
Transition Table Used In the Dragon Book

Problem: Translate $(a \mid b)^* abb$ into a DFA.

Solution:



NFA State	DFA State	a	b
{0,1,2,4,7}	A	B	C
{1,2,3,4,6,7,8}	B	B	D
{1,2,4,5,6,7}	C	B	C
{1,2,4,5,6,7,9}	D	B	E
{1,2,4,5,6,7,10}	E	B	C



Subset Construction

An DFA can be exponentially larger than the corresponding NFA.

n states versus 2^n

Tools often try to strike a balance between the two representations.

Lexical Analysis with Ocamllex

Constructing Scanners with Ocamllex

$$\boxed{\text{scanner.mll}} \xrightarrow[\text{(subset construction)}]{\text{ocamllex}} \boxed{\text{scanner.ml}}$$

An example:

scanner.mll

```
{ open Parser }

rule token =
  parse [' ' '\t' '\r' '\n'] { token lexbuf }
      | '+'                  { PLUS }
      | '-'                  { MINUS }
      | '*'                  { TIMES }
      | '/'                  { DIVIDE }
      | ['0'-'9']+ as lit    { LITERAL(int_of_string lit) }
      | eof                  { EOF }
```

Ocamllex Specifications

```
{
  (* Header: verbatim OCaml code; mandatory *)
}

(* Definitions: optional *)
let ident = regexp
let ...

(* Rules: mandatory *)
rule entrypoint1 [arg1 ... argn] =
  parse pattern1 { action (* OCaml code *) }
      | ...
      | patternn { action }
and entrypoint2 [arg1 ... argn]} =
  ...
and ...

{
  (* Trailer: verbatim OCaml code; optional *)
}
```

Patterns (In Order of Decreasing Precedence)

Pattern	Meaning
'c'	A single character
_	Any character (underline)
eof	The end-of-file
"foo"	A literal string
['1' '5' 'a'-'z']	“1,” “5,” or any lowercase letter
[^ '0'-'9']	Any character except a digit
(*pattern*)	Grouping
identifier	A pattern defined in the let section
pattern *	Zero or more *patterns*
pattern +	One or more *patterns*
pattern ?	Zero or one *patterns*
$pattern_1$ $pattern_2$	$pattern_1$ followed by $pattern_2$
$pattern_1 \mid pattern_2$	Either $pattern_1$ or $pattern_2$
pattern as *id*	Bind the matched pattern to variable *id*

An Example

```
{ type token = PLUS | IF | ID of string | NUM of int }
let letter = ['a'-'z' 'A'-'Z']
let digit = ['0'-'9']

rule token =
 parse [' ' '\n' '\t'] { token lexbuf } (* Ignore whitespace *)

     | '+' { PLUS }                     (* A symbol *)

     | "if" { IF }                      (* A keyword *)
                                        (* Identifiers *)
     | letter (letter | digit | '_')* as id { ID(id) }
                                        (* Numeric literals *)
     | digit+ as lit { NUM(int_of_string lit) }

     | "/*" { comment lexbuf }          (* C-style comments *)

and comment =
  parse "*/" { token lexbuf } (* Return to normal scanning *)
      | _ { comment lexbuf }  (* Ignore other characters *)
```

Nested Comments

```
{ type token = PLUS | ID of string | NUM of int }

let letter = ['a'-'z' 'A'-'Z']
let digit = ['0'-'9']

rule token =
 parse [' ' '\n' '\t'] { token lexbuf } (* Ignore whitespace *)

     | '+' { PLUS }                     (* A symbol *)

     | letter (letter | digit | '_')* as id { ID(id) }
     | digit+ as lit { NUM(int_of_string lit) }

     | "/*" { comment 0 lexbuf }          (* C-style comments *)

and comment level =
  parse "*/" { if level == 0 then token lexbuf
        else comments (level - 1) lexbuf }
      | "/*" { comment (level + 1) lexbuf }
      | _ { comment level lexbuf }  (* Ignore other characters *)
```

Free-Format Languages

Typical style arising from scanner/parser division

Program text is a series of tokens possibly separated by whitespace and comments, which are both ignored.

- keywords (if while)
- punctuation (, (+)
- identifiers (foo bar)
- numbers (10 -3.14159e+32)
- strings ("A String")

Free-Format Languages

Java C C++ C# Algol Pascal

Some deviate a little (e.g., C and C++ have a separate preprocessor)

But not all languages are free-format.

Python

The Python scripting language groups with indentation

```
i = 0
while i < 10:
    i = i + 1
    print i     # Prints 1, 2, ..., 10

i = 0
while i < 10:
    i = i + 1
print i         # Just prints 10
```

This is succinct, but can be error-prone.

How do you wrap a conditional around instructions?

Syntax and Language Design

Does syntax matter? Yes and no

More important is a language's *semantics*—its meaning.

The syntax is aesthetic, but can be a religious issue.

But aesthetics matter to people, and can be critical.

Verbosity does matter: smaller is usually better.

Too small can be problematic: APL is a succinct language with its own character set.

There are no APL programs, only puzzles.

Syntax and Language Design

Some syntax is error-prone. Classic FORTRAN example:

```
DO 5 I = 1,25   ! Loop header (for i = 1 to 25)
DO 5 I = 1.25   ! Assignment to variable DO5I
```

Trying too hard to reuse existing syntax in C++:

```
vector< vector<int> > foo;
vector<vector<int>> foo; // Syntax error
```

C distinguishes $>$ and $>>$ as different operators.

Bjarne Stroustrup tells me they have finally fixed this.