REGULAR LANGUAGES
AND
LEXICAL ANALYSIS

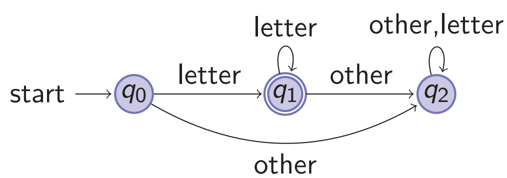
a.y. 2022-2023

Recognizing languages

- $\mathcal{L} = \{a^n b^n \mid n > 0\}$ is a free language
- Stack seem ideal to recognize words in $\mathcal{L}$
- Read the string one symbol at a time: push *a*s onto stack; pop one symbol for each *b* and reject further *a*s; check emptiness
- Do we really need all that "counting" to generate, e.g., arbitrary strings over the English alphabet?

Recognizing languages
ARBITRARY STRINGS OVER THE ALPHABET

- Do we really need all that "counting" to generate, e.g., arbitrary strings over the alphabet?
- $S \to a \mid b \mid \dots \mid z \mid aS \mid bS \mid \dots \mid zS$
- A state machine is enough



Regular languages

- Regular grammars: free grammars with productions of the form
  - $A \to a$
  - $A \to aB$
  - $A \to \epsilon$
- Regular expressions
- Nondeterministic finite state automata
- Deterministic finite state automata
- At the basis of lexical analysis

Regular expressions

- Fix an alphabet $\mathcal{A}$, and a number of operators
- Define regular expressions inductively
  - Base:
    - Every $a \in \mathcal{A}$ is a regular expression
    - ϵ is a regular expression
  - Step: If r_1 and r_2 are regular expressions then
    - (Alternation) $r_1 \mid r_2$ is a regular expressions
    - (Concatenation) $r_1 \cdot r_2$ is a regular expressions (written $r_1 r_2$)
    - (Kleene star) ${r_1}^*$ is a regular expressions
    - (Parentheses) (r_1) is a regular expression

Regular expressions

DENOTED LANGUAGE

- Given a regular expression r over $\mathcal{A}$, the **language denoted** by r, written $\mathcal{L}(r)$, is also inductively defined on the structure of r
  - Base:
    - $\mathcal{L}(a) = \{a\}$ for every $a \in \mathcal{A}$
    - $\mathcal{L}(\epsilon) = \{\epsilon\}$
  - Step:
    - If $r = r_1 \mid r_2$ then $\mathcal{L}(r) = \mathcal{L}(r_1) \cup \mathcal{L}(r_2)$
    - If $r = r_1 r_2$ then $\mathcal{L}(r) = \{w_1 w_2 \mid w_1 \in \mathcal{L}(r_1) \text{ and } w_2 \in \mathcal{L}(r_2)\}$
    - If $r = {r_1}^*$ then
      $\mathcal{L}(r) = \{\epsilon\} \cup \{w_1 w_2 \ldots w_k \mid k \geq 1 \text{ and } \forall i : 1 \leq i \leq k. w_i \in \mathcal{L}(r_1)\}$
    - If $r = (r_1)$ then $\mathcal{L}(r) = \mathcal{L}(r_1)$

Regular expressions
CONVENTIONS

- Kleene star has highest precedence, is left associative
- Concatenation has second highest precedence, is left associative
- Alternation has lowest precedence, is left associative

- Then $(a \mid bc^*)$ means
  - $(a \mid b(c^*))$
  - $(a \mid (b(c^*)))$

Regular expressions
EXAMPLES

- $\mathcal{L}(a \mid b) = \{a, b\}$
- $\mathcal{L}((a \mid b)(a \mid b)) = \{aa, ab, ba, bb\}$
- $\mathcal{L}(a^*) = \{a^n \mid n \geq 0\}$
- $\mathcal{L}(a \mid a^*b) = \{a\} \cup \{a^n b \mid n \geq 0\}$

Regular expressions
EXAMPLES

- $(a \mid b \mid \ldots \mid z)(a \mid b \mid \ldots \mid z)^*$ denotes the set of words over the alphabet
- $(0 \mid 1)^*0$ denotes all the even binary numbers
- $b^*(abb^*)^*(a \mid \epsilon)$ denotes the set of words over $\{a, b\}$ with no consecutive occurrences of a
- $(a \mid b)^*aa(a \mid b)^*$ denotes the set of words over $\{a, b\}$ with consecutive occurrences of a

Finite states automata

- Finite state automata are used to decide whether a word belongs to the language denoted by a regular expression
- Nondeterministic Finite state Automata (NFA)
- Deterministic Finite state Automata (DFA)

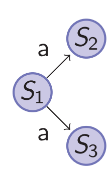
Nondeterministic finite state automata

- An NFA is a tuple $(S, \mathcal{A}, \mathrm{move}_n, s_0, F)$
- Where
  - S is a set of states
  - $\mathcal{A}$ is an alphabet with $\epsilon \notin \mathcal{A}$
  - $s_0 \in S$ is the **initial** state
  - $F \subseteq S$ is the set of **final** (or **accepting**) states
  - $\mathrm{move}_n : S \times (\mathcal{A} \cup \{\epsilon\}) \to 2^S$ is the transition function

Nondeterministic finite state automata

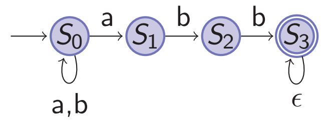
GRAPHICAL REPRESENTATION

- $(S, \mathcal{A}, \mathrm{move}_n, s_0, F)$ is represented as a directed graph
- Where
  - Nodes represent states
  - The initial state is identified by an incoming arrow
  - Final states are identified by a double circle
  - Edges represent the transition function
  - E.g., suppose $\mathrm{move}_n(S_1, a) = \{S_2, S_3\}$, then



Nondeterministic finite state automata
EXAMPLE

An NFA



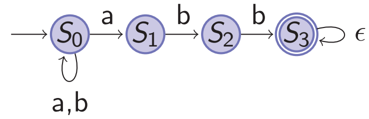
Tabular representation of its transition function

	ϵ	a	b
S_0	$\emptyset$	$\{S_0, S_1\}$	$\{S_0\}$
S_1	$\emptyset$	$\emptyset$	$\{S_2\}$
S_2	$\emptyset$	$\emptyset$	$\{S_3\}$
S_3	$\{S_3\}$	$\emptyset$	$\emptyset$

Nondeterministic finite state automata
ACCEPTED LANGUAGES

- The NFA $\mathcal{N}$ **accepts** (or **recognizes**) w iff there exists **at least** a path spelling w from its initial state to one of its final states
- Recall
  - $\epsilon\epsilon$ spells ϵ
  - $\boldsymbol{a}\epsilon$ spells $\boldsymbol{a}$
  - $\epsilon\boldsymbol{a}$ spells $\boldsymbol{a}$
- The language **accepted** (or **recognized**) by $\mathcal{N}$, written $\mathcal{L}(\mathcal{N})$, is the set of all the strings accepted by $\mathcal{N}$

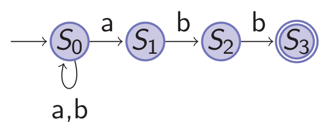
Nondeterministic finite state automata
EXAMPLE



Accepted language

$$\mathcal{L}((a \mid b)^* abb)$$

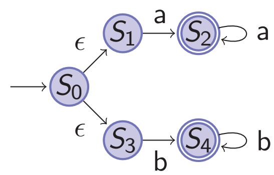
Nondeterministic finite state automata
EXAMPLE



Accepted language

$$\mathcal{L}((a \mid b)^* abb)$$

Nondeterministic finite state automata
EXAMPLE



Accepted language

$$\mathcal{L}(aa^* \mid bb^*)$$

Thompson's construction

An algorithm to construct an NFA $\mathcal{N}$ from a regular expression r in such a way that

$$\mathcal{L}(\mathcal{N}) = \mathcal{L}(r)$$

Thompson's construction

The construction is based on the inductive definition of regular expressions

- Base: r is either ϵ or a symbol of the alphabet
  - Define an NFA to recognize $\mathcal{L}(\epsilon)$
  - Define an NFA to recognize $\mathcal{L}(a)$
- Step: r is either $r_1 \mid r_2$, or $r_1 r_2$, or $r_1{}^*$, or (r_1)
  - Given NFAs $\mathcal{N}_1$ and $\mathcal{N}_2$ such that $\mathcal{L}(\mathcal{N}_i) = \mathcal{L}(r_i)$ for $i = 1, 2$
    - Define an NFA to recognize $\mathcal{L}(r_1 \mid r_2)$
    - Define an NFA to recognize $\mathcal{L}(r_1 r_2)$
    - Define an NFA to recognize $\mathcal{L}(r_1{}^*)$
    - Define an NFA to recognize $\mathcal{L}((r_1))$

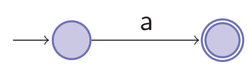
Thompson's construction

MAIN FEATURES

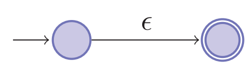
- Every step of the construction introduces 2 new states at most
  - The generated NFA has $2k$ states at most, where k is the number of symbols and of operators in the regular expression
- In every intermediate NFA there is
  - Exactly one final state
  - No edge incoming in the initial state
  - No edge outgoing from the final state

Thompson's construction
BASE

$r = a$

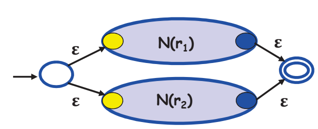


$r = \epsilon$

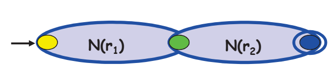


Thompson's construction
STEP

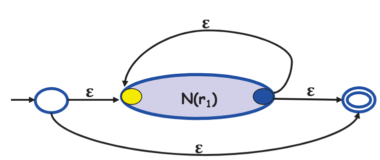
$r = r_1 \mid r_2$



$r = r_1 r_2$



$r = r_1^*$



$r = (r_1)$

Thompson's construction
COMPLEXITY

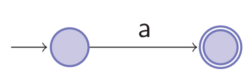
- Constructs an NFA with n nodes and m edges
- Every step adds at most 2 states and 4 edges
- Every step has constant time
- There are $|r|$ steps

- Then:
  - **Space:** $n + m$ is $\mathcal{O}(|r|)$
  - **Time:** $\mathcal{O}(|r|)$

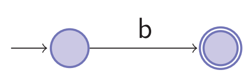
Thompson's construction
EXAMPLE OF APPLICATION

Take $r = (a \mid b)^* abb$

$a = r_1$



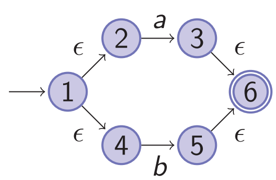
$b = r_2$



Thompson's construction
EXAMPLE OF APPLICATION

$$a \mid b = r_1 \mid r_2 = r_3$$

Apply Thompson's construction for alternation to the automata for r_1 and r_2, and get the automaton for r_3

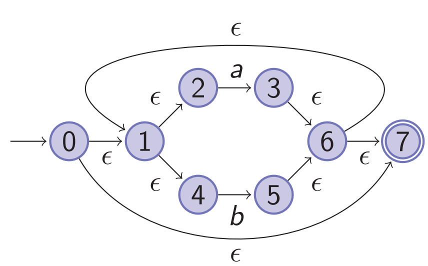


FORMAL LANGUAGES AND COMPILERS Paola Quaglia, 2022

Thompson's construction
EXAMPLE OF APPLICATION

$$(a \mid b) = (r_3) = r_4 \qquad (a \mid b)^* = {r_4}^* = r_5$$

Apply Thompson's construction for parentheses and for Kleene star to the automaton for r_3, and get the automaton for r_5

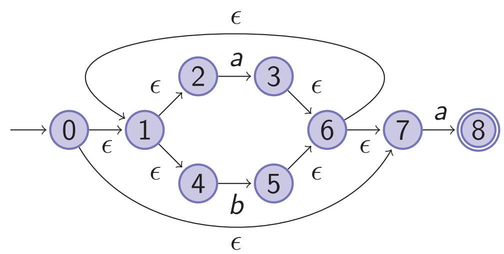


Thompson's construction

EXAMPLE OF APPLICATION

$$(a \mid b)^* a = r_5 r_1 = r_6$$

Apply Thompson's construction for concatenation to the automata for r_5 and for r_1, and get the automaton for r_6

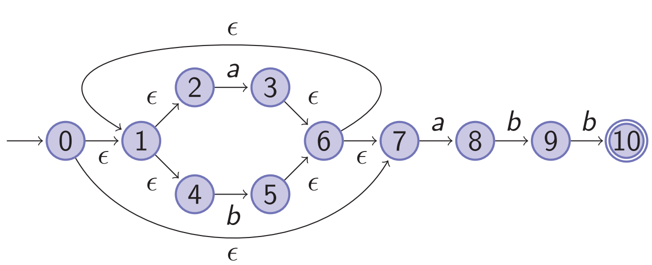


Thompson's construction

EXAMPLE OF APPLICATION

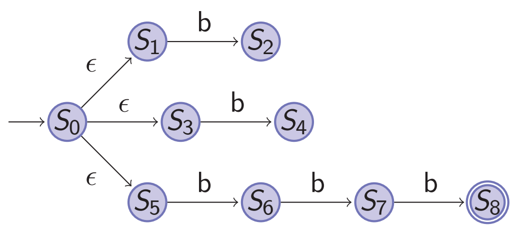
$$r = (a \mid b)^* abb$$

In two more steps



Simulation of NFAs

- Given a word w and an NFA $\mathcal{N}$ decide whether $w \in \mathcal{L}(\mathcal{N})$
- Let $w = bbb$



- Reading bbb from S_0 is the same as reading it from any of the states in $\{S_0, S_1, S_3, S_5\}$

Simulation of NFAs

ϵ-CLOSURE

- Let $(S, \mathcal{A}, \mathrm{move}_n, s_0, F)$ be an NFA, t be a state in S, and T be a subset of states
- $\epsilon\text{-closure}(\{t\})$ is the set of states in S which are reachable from t by zero or more ϵ-transitions
- $\epsilon\text{-closure}(T) = \bigcup_{t \in T} \epsilon\text{-closure}(\{t\})$

Simulation of NFAs
COMPUTATION OF ϵ-CLOSURE

Data structures:

- Stack
- Boolean array "alreadyOn" of size $|S|$ to check in constant time whether a state t in onto the stack
- Bidimensional array to record move_n. Every entry (t, x) is a linked list containing all the states in $\mathrm{move}_n(t, x)$

Simulation of NFAs
COMPUTATION OF ϵ-CLOSURE

```
foreach i=1,..., |S| do  alreadyOn[i] = false ;
function closure(t,stack)
    push t onto stack ;
    alreadyOn[t] = true ;
    foreach u ∈ move_n(t, ϵ) do
        if not alreadyOn[u] then
            closure (u,stack) ;
```

Simulation of NFAs

COMPLEXITY OF ϵ-CLOSURE

1 push t onto stack;
2 set alreadyOn[t] bit;
3 find next $u \in \mathrm{move}_n(t, \epsilon)$;
4 test alreadyOn[u] bit;

- Each of them takes constant time
- How many times are they repeated?

Simulation of NFAs

COMPLEXITY OF ϵ-CLOSURE

1 push t onto stack;
2 set alreadyOn[t] bit;
3 find next $u \in \mathrm{move}_n(t, \epsilon)$;
4 test alreadyOn[u] bit;

- Line1 & Line2: Executed at each invocation of closure (either initial or recursive)
  - Every state goes onto stack at most once (alreadyOn bit initially false, then set to true, never changed again)
- In aggregate, assuming that the NFA has n states and m edges, $\mathcal{O}(n)$

Simulation of NFAs

COMPLEXITY OF ϵ-CLOSURE

```
1 push t onto stack;
2 set alreadyOn[t] bit;
3 find next u ∈ move_n(t, ϵ);
4 test alreadyOn[u] bit;
```

- Line3 & Line4: Executed at each invocation of closure for all $u \in \mathrm{move}_n(t, \epsilon)$
  - In the worst case every state goes onto the stack, and every state has at least an ϵ-transition
- In aggregate, assuming that the NFA has n states and m edges, $\mathcal{O}(m)$

Simulation of NFAs

COMPLEXITY OF ϵ-CLOSURE

```
function closure(t,stack)
    push t onto stack ;
    alreadyOn[t] = true ;
    foreach u ∈ move_n(t, ϵ) do
        if not alreadyOn[u] then
            closure (u,stack) ;
```

$$\mathcal{O}(n + m)$$

Simulation of NFAs

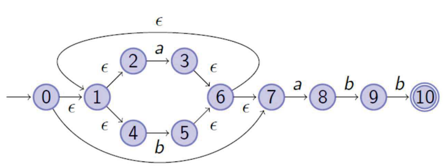
ALGORITHM

input : NFA $\mathcal{N} = (S, \mathcal{A}, \mathrm{move}_n, s_0, F)$, w\$
output : "yes" if w$\in \mathcal{L}(\mathcal{N})$, "no" otherwise
states $= \epsilon$-closure$(\{s_0\})$;
symbol = nextchar() ;
while *symbol* $\neq$ \$ **do**
  states $= \epsilon$-closure$(\bigcup_{t \in \text{states}} \mathrm{move}_n(t, \text{symbol}))$;
  symbol = nextchar() ;
if *states* $\cap$ $F \neq \emptyset$ **then**
  return "yes" ;
else
  return "no" ;

Simulation of NFAs

EXAMPLE

w=ababb



states	symbol	U_t move(t,symbol)	ε-closure
T0 = {0,1,2,4,7}	a	{3,8}	{1,2,3,4,6,7,8}
T1 = {1,2,3,4,6,7,8}	b	{5,9}	{1,2,4,5,6,7,9}
T2 ={1,2,4,5,6,7,9}	a	{3,8}	T1
T1	b		T2
T2	b	{5,10}	{1,2,4,5,6,7,10}
T3={1,2,4,5,6,7,10}	\$		

Simulation of NFAs
COMPLEXITY

Dominant

while *symbol* $\neq$ \$ **do**
L1 states $= \epsilon\text{-closure}(\bigcup_{t \in \text{states}} \text{move}_n(t, \text{symbol}))$;
symbol $=$ nextchar() ;

Simulation of NFAs
COMPLEXITY

Data structures:

- Two stacks for states:
  - currentStack for the current states ("states" on the right-side of the assignment at Line L1)
  - nextStack for the next states ("states" on the left-side of the assignment at Line L1)
- Boolean array "alreadyOn" of size $|S|$ to check in constant time whether a state in onto the stack
- Bidimensional array to record move_n. Every entry (t, x) is a linked list containing all the states in $\text{move}_n(t, x)$

Simulation of NFAs

COMPLEXITY, LINE L1

```
foreach t on currentStack do
    foreach u ∈ move_n(t, symbol) do
        if not alreadyOn[u] then
            closure(u,nextStack);
    pop t from currentStack;
foreach s on nextStack do
    pop s from nextStack ;
    push s on currentStack ;
    alreadyOn[s] = false ;
```


Simulation of NFAs

COMPLEXITY, LINE L1

```
foreach t on currentStack do
    ⋮
    ;                               /* populate nextStack */
foreach s on nextStack do
    ⋮
    ;                   /* swap nextStack with currentStack */
```

Simulation of NFAs

COMPLEXITY

- Assume that the NFA has n states and m edges
- For each while-cycle
  - Populating nextStack is $\mathcal{O}(n+m)$
  - Swapping the stacks is $\mathcal{O}(n)$
- One while-cycle costs $\mathcal{O}(n+m)$
- The simulation of w costs $\mathcal{O}(|w|(n+m))$
- If the NFA results from Thompson's construction, $(n+m)$ is $\mathcal{O}(|r|)$, then simulating w is $\mathcal{O}(|w||r|)$

Wrap-up: Accepting regular languages by NFAs

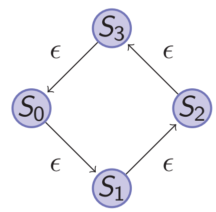
- Take a regular expression r
- Apply Thomson's construction, get NFA $\leftarrow \mathcal{O}(|r|)$
- Simulate NFA $\leftarrow \mathcal{O}(|w||r|)$
- Overall: $\mathcal{O}(|w||r|)$

On ϵ-closure, again

Let $(S, \mathcal{A}, \mathrm{move}_n, s_0, F)$ be an NFA, and let $M \subseteq S$.
Then $\epsilon\text{-closure}(M)$ is **the least** set $X \subseteq S$ that is a solution to the set equation

$$X = M \cup \{N' \mid N \in X \text{ and } N' \in \mathrm{move}_n(N, \epsilon)\}$$

- For example



- $\epsilon\text{-closure}(\{S_0\}) = \{S_0, S_1, S_2, S_3\}$
- Start with S_0 in the set
- Take S_1 because $S_1 \in \mathrm{move}_n(S_0, \epsilon)$
- ...

Fixed Point

- The set equation

$$X = M \cup \{N' \mid N \in X \text{ and } N' \in \mathrm{move}_n(N, \epsilon)\}$$

- Is an instance of the general form of set equation

$$X = f(X)$$

Fixed Point

- If
  - $f : 2^D \to 2^D$ for some finite set D, and
  - f is monotonic, i.e. $X \subseteq Y$ implies $f(X) \subseteq f(Y)$
- Then there is a precise technique to solve the equation $X = f(X)$

Fixed Point

THEOREM
Let S be a finite set, and let $f : 2^S \to 2^S$ be monotonic.
Then $\exists m \in \mathbb{N}$ such that the unique minimal solution to the set equation $X = f(X)$ is $f^m(\emptyset)$

Fixed Point
PROOF

- Show that $m \in \mathbb{N}$ exists such that $f^m(\emptyset)$ is a solution to $X = f(X)$
- Show that $f^m(\emptyset)$ is the unique minimal solution

Fixed Point
PROOF

$m \in \mathbb{N}$ exists such that $f^m(\emptyset)$ is a solution to $X = f(X)$

- $\emptyset \subseteq f(\emptyset)$
- Then, by f monotonic, $f(\emptyset) \subseteq f^2(\emptyset)$
- Then, by induction, $f^i(\emptyset) \subseteq f^{i+1}(\emptyset)$ forall $i \in \mathbb{N}$
- Then we have a chain $\emptyset \subseteq f^1(\emptyset) \subseteq f^2(\emptyset) \subseteq f^3(\emptyset) \ldots$
- Then, by 2^S finite, the sets in the chain cannot be all different
- Then for some m it must be $f^m(\emptyset) = f^{m+1}(\emptyset) = f(f^m(\emptyset))$
- Then $f^m(\emptyset)$ is a solution to $X = f(X)$

Fixed Point
PROOF

$f^m(\emptyset)$ is the unique minimal solution

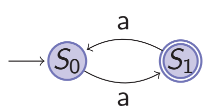
- Suppose another solution to $X = f(X)$ exists, say A.
- Then, by hypothesis, $A = f(A)$
- Then, $A = f(A)$, $A = f(A) = f^2(A) = \ldots = f^m(A)$
- Then, by $\emptyset \subseteq A$ and f monotonic, $f^m(\emptyset) \subseteq f^m(A)$
- Then, by $f^m(A) = A$, $f^m(\emptyset) \subseteq A$
- Then $f^m(\emptyset)$ is the unique minimal solution to $X = f(X)$

Deterministic finite state automata

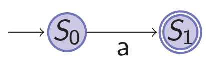
- NFA: $(S, \mathcal{A}, \mathrm{move}_n, s_0, F)$
  where $\mathrm{move}_n : S \times (\mathcal{A} \cup \{\epsilon\}) \to 2^S$
- DFA: $(S, \mathcal{A}, \mathrm{move}_d, s_0, F)$
  where $\mathrm{move}_d : S \times \mathcal{A} \to S$
- In any DFA
  - There is no ϵ-transition
  - If move_d is **total**, then from every state there is **exactly one** a-transition for every $a \in \mathcal{A}$
  - If move_d is **partial**, then from every state there is **at most one** a-transition for every $a \in \mathcal{A}$

Deterministic finite state automata

- Take the alphabet $\{a\}$
- An example of DFA with total transition function is



- An example of DFA with partial transition function is



Simulation of DFAs

ACCEPTED LANGUAGES

The language recognized by a DFA $\mathcal{D}$, denoted by $\mathcal{L}(\mathcal{D})$, is the set of words w such that

- Either there is a path spelling $w = a_1 \cdots a_k$ with $k >= 1$ from the initial state of $\mathcal{D}$ to some of its final states
- Or the initial state is also final and $w = \epsilon$

Simulation of DFAs
TOTAL TRANSITION FUNCTION

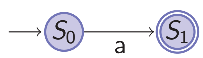
- Answer the question "$w \in \mathcal{L}(\mathcal{D})$?"
- Starting from the initial state, follow the path spelling w
- If the reached state is final, then return "yes"
- Otherwise return "no"

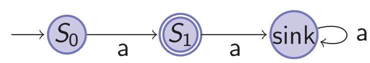
Simulation of DFAs
PARTIAL TRANSITION FUNCTION

- Answer the question "$w \in \mathcal{L}(\mathcal{D})$?"
- Starting from the initial state, begin following the path spelling $w = a_1 \cdots a_k$
- If for some a_i there is no target state, then return "no"
- If w is over and the reached state is final, then return "yes"
- Otherwise return "no"

Deterministic finite state automata
PARTIAL VS TOTAL TRANSITION FUNCTION

- Let $\mathcal{D}$ be a DFA with partial transition function
- Can you define a DFA $\mathcal{D}'$ with total transition function such that $\mathcal{L}(\mathcal{D}') = \mathcal{L}(\mathcal{D})$?
- Use a **sink**
  - Add a sink to the states of $\mathcal{D}$
  - Let the sink be the target of all the undefined transitions
  - For every symbol in the alphabet, add the sink with a self-loop





Subset construction

Given the NFA $\mathcal{N}$ construct a DFA $\mathcal{D}$ such that $\mathcal{L}(\mathcal{D}) = \mathcal{L}(\mathcal{N})$

Idea: Use ϵ-closure to map subsets of states of the NFA into one single state of the DFA

Subset Construction

- This is the first instance of a construction we will see again later in the course
- Define the initial state of the DFA
- Populate the collection of all its states while defining the transition function

```
foreach state T already collected do
    foreach symbol a in the alphabet do
        compute the plausible target of the a-transition from T ;
        call it T';
        if T' is already in the collection then
            nothing else to do ;
        else
            add T' to the collection ;
```


Subset Construction

input : NFA $\mathcal{N} = (S, \mathcal{A}, \text{move}_n, S_0, F)$
output : DFA $\mathcal{D} = (R, \mathcal{A}, \text{move}_d, T_0, E)$ such that $\mathcal{L}(\mathcal{D}) = \mathcal{L}(\mathcal{N})$

$T_0 = \epsilon\text{-closure}(\{S_0\})$;
$R = \{T_0\}$;
set T_0 as unmarked ;
while *some* $T \in R$ *is unmarked* **do**
    mark T ;
    foreach $a \in \mathcal{A}$ **do**
        $T' = \epsilon\text{-closure}(\bigcup_{t \in T} \text{move}_n(t, a))$;
        if $T' \neq \emptyset$ **then**
            $\text{move}_d(T, a) = T'$;
            if $T' \notin R$ **then**
                add T' to R ;
                set T' as unmarked ;
foreach $T \in R$ **do**
    if $(T \cap F) \neq \emptyset$ **then** set $T \in E$;

Subset Construction
COMPLEXITY

Dominant

while *some* $T \in R$ *is unmarked* **do**
  $\ldots$;
  foreach $a \in \mathcal{A}$ **do**
    $T' = \epsilon\text{-closure}(\bigcup_{t \in T} \text{move}_n(t, \mathsf{a}))$;
    $\vdots$

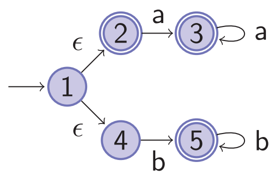
Subset Construction
COMPLEXITY

while *some* $T \in R$ *is unmarked* **do**
  $\ldots$;
  foreach $a \in \mathcal{A}$ **do**
    $T' = \epsilon\text{-closure}(\bigcup_{t \in T} \text{move}_n(t, \mathsf{a}))$;
    $\vdots$

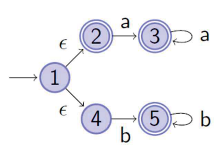
- Suppose the NFA has n states and m edges, and the DFA has n_d states
- **while** is repeated n_d times
- **foreach** is repeated $|\mathcal{A}|$ times
- $\epsilon\text{-closure}(\bigcup_{t \in T} \text{move}_n(t, \mathsf{a}))$ costs $\mathcal{O}(n + m)$
- Then the subset construction is $\mathcal{O}(n_d \cdot |\mathcal{A}| \cdot (n + m))$
- The question is: how big can n_d be?

Subset construction
EXAMPLE OF APPLICATION

From the NFA



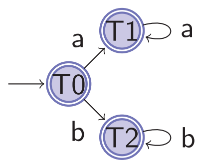
Subset construction
EXAMPLE OF APPLICATION



	a	b
T0 = ε-closure({1}) = {1,2,4} % initial, final	ε-closure({3}) = T1	ε-closure({5}) = T2
T1 = {3} % final	ε-closure({3}) = T1	---
T2 = {5} % final	---	ε-closure({5}) = T2

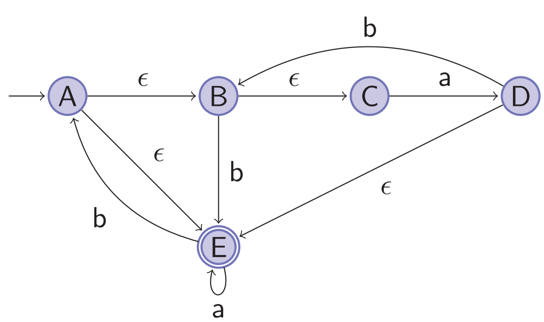
Subset construction
EXAMPLE OF APPLICATION

Get the DFA



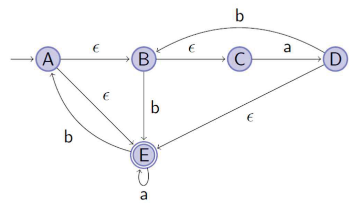
Subset construction
EXAMPLE OF APPLICATION

From the NFA



Subset construction

EXAMPLE OF APPLICATION

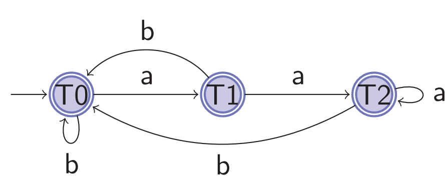


	a	b
T0 = ε-closure({A}) = {A,B,C,E} % initial, final	ε-closure({D,E}) = T1	ε-closure({A,E}) = {A,B,C,E} = T0
T1 = {D,E} % final	ε-closure({E}) = T2	ε-closure({A,B}) = {A,B,C,E} = T0
T2 = {E} % final	ε-closure({E}) = T2	ε-closure({A}) = T0

Subset construction

EXAMPLE OF APPLICATION

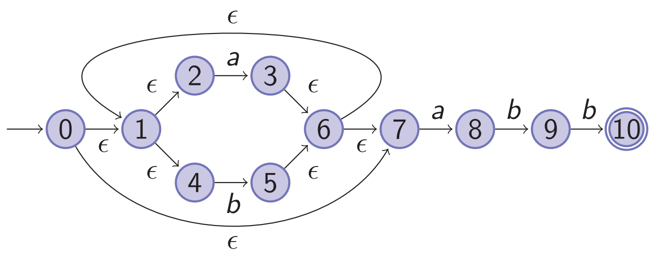
Get the DFA



Subset construction

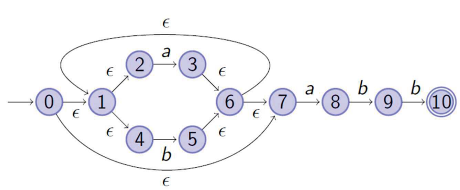
EXAMPLE OF APPLICATION

From the NFA



Subset construction

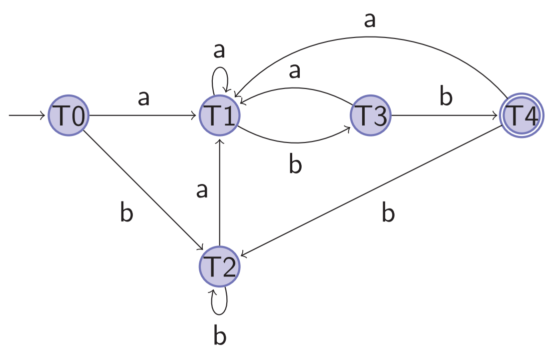
EXAMPLE OF APPLICATION



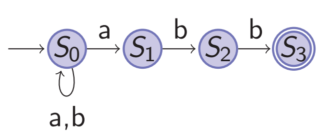
	a	b
T0 = ε-closure({0}) = {0,1,2,4,7} % initial	ε-closure({3,8}) = T1	ε-closure({5}) = T2
T1 = {1,2,3,4,6,7,8}	ε-closure({3,8}) = T1	ε-closure({5,9}) = T3
T2 = {1,2,4,5,6,7}	ε-closure({3,8}) = T1	ε-closure({5}) = T2
T3 = {1,2,4,5,6,7,9}	ε-closure({3,8}) = T1	ε-closure({5,10}) =T4
T4 = {1,2,4,5,6,7,10} % final	ε-closure({3,8}) = T1	ε-closure({5}) = T2

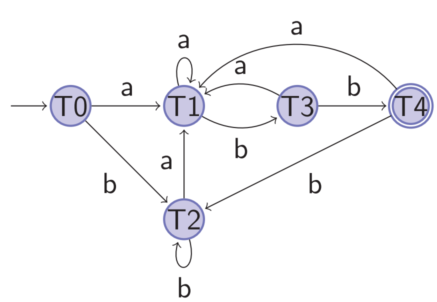
Subset construction
EXAMPLE OF APPLICATION

Get the DFA



Subset construction





Both automata recognize $\mathcal{L}((a \mid b)^* abb)$

Can we get a smaller deterministic structure accepting the same language?

DFA minimization

Given the DFA $\mathcal{D}$ construct a minimal DFA $\mathcal{D}'$ such that
$\mathcal{L}(\mathcal{D}') = \mathcal{L}(\mathcal{D})$

DFA minimization

INTUITION

Partition the states of the DFA into equivalent classes

Two states s and t are equivalent iff
for every x
the simulation of x from s is successfull iff
the simulation of x from t is successfull

DFA minimization
STATE EQUIVALENCE

Let $\mathcal{D} = (S, \mathcal{A}, \mathrm{move}_d, s_0, F)$ be a DFA with **total** transition function
Then $s, t \in S$ are **equivalent** iff the following holds

$\mathrm{move}_d{}^*(s, x) \in F$ *iff* $\mathrm{move}_d{}^*(t, x) \in F$ *for every* $x \in \mathcal{A}^*$

where the multi-step transition function $\mathrm{move}_d{}^*$ is defined by induction on the length of strings

- $\mathrm{move}_d{}^*(s, \epsilon) = s$
- $\mathrm{move}_d{}^*(s, wa) = \mathrm{move}_d(\mathrm{move}_d{}^*(s, w), a)$

DFA minimization
PARTITION REFINEMENT

- Partition the states into blocks (subsets of states)
- Start with the two blocks
  - $B_1 = F$
  - $B_2 = S \setminus F$
  - Why this choice for the initial partition?
  - $s \in B_1$ and $t \in B_2$ are not equivalent because $\mathrm{move}_d{}^*(s, \epsilon) \in F$ and $\mathrm{move}_d{}^*(t, \epsilon) \notin F$

DFA minimization
PARTITION REFINEMENT

- Check whether the blocks contain equivalent states
- If not so, refine the blocks by splitting them further
- Repeat the checking-refining steps up to the point that the partition cannot be further refined

DFA minimization
PARTITION REFINEMENT: SPLITTING BLOCKS

- If all the states in $B_i = \{s_1, \ldots, s_k\}$ are equivalent, then for every $a \in \mathcal{A}$ the target states of the a-transition from $s_1, \ldots, s_k$ should belong to the same block
- The block B_i can be split wrt (a, B_j) if for some $s, t \in B_i$

  $\mathrm{move}_d(s, a) \in B_j$ *and* $\mathrm{move}_d(t, a) \notin B_j$
- Splitting the block B_i wrt (a, B_j) amounts to replacing B_i by the two blocks
  - $\{s \in B_i \mid \mathrm{move}_d(s, a) \in B_j\}$
  - $\{s \in B_i \mid \mathrm{move}_d(s, a) \notin B_j\}$

DFA minimization

PARTITION REFINEMENT

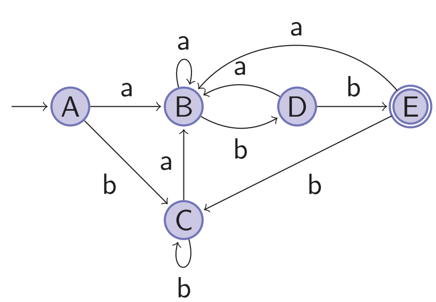
$B_1 = F$;
$B_2 = S \setminus F$;
$P = \{B_1, B_2\}$;
while *Some $B_i \in P$ can be split wrt some (a, B_j)* **do**
  Update P by removing B_i and adding the two blocks
  $\{s \in B_i \mid \mathrm{move}_d(s, a) \in B_j\}$ and
  $\{s \in B_i \mid \mathrm{move}_d(s, a) \notin B_j\}$

DFA minimization

PARTITION REFINEMENT

Block B_i can be split wrt (a, B_j) if for some $s, t \in B_i$

- $\mathrm{move}_d(s, a) \in B_j$ and $\mathrm{move}_d(t, a) \notin B_j$

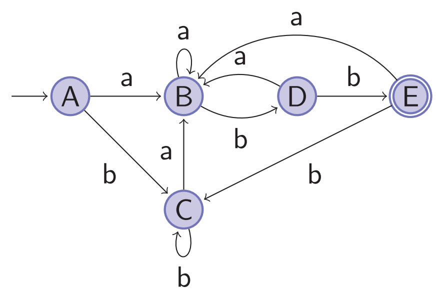


Initial Partition: $B_1 = \{E\}$ $B_2 = \{A, B, C, D\}$

- $\mathrm{move}_d(D, b) = E \in B_1$ and $\mathrm{move}_d(A, b) = C \notin B_1$
- Then we can split B_2 wrt (b, B_1)

DFA minimization

PARTITION REFINEMENT



Init	$B_1 = \{E\};\ B_2 = \{A, B, C, D\}$
split B_2	$B_1 = \{E\};\ B_{21} = \{D\};\ B_{22} = \{A, B, C\}$
split B_{22}	$B_1 = \{E\};\ B_{21} = \{D\};\ B_{221} = \{B\};\ B_{222} = \{A, C\}$

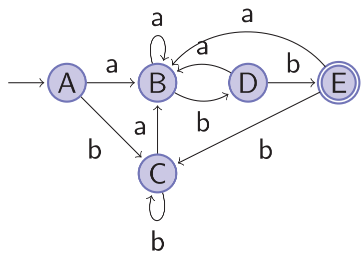
DFA Minimization

- Run partition refinement over a DFA with **total** transition function
- Each block is *temporarily* considered a state of the min-DFA
- The initial state of the min-DFA is the one represented by the block containing s_0
- The final states of the min-DFA are those represented by blocks containing final states of the original DFA
- The transition function of the min-DFA is derived from the original move_d as follows:

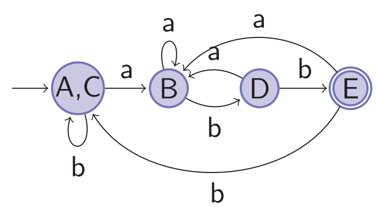
  If $s \in B_i$ and $\mathrm{move}_d(s, a) \in B_j$, then there is an a-transition in the min-DFA from the state represented by B_i to the state represented by B_j
- If any of the temporary state is dead, remove it, and remove all the transitions to/from it
- **Observe:** The min-DFA can have **partial** transition function

DFA minimization

PARTITION REFINEMENT

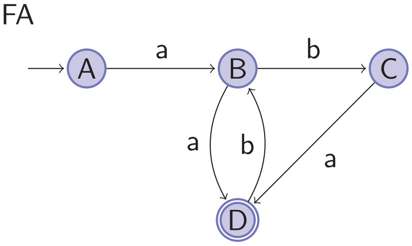


$\{E\}$; $\{D\}$; $\{B\}$; $\{A, C\}$

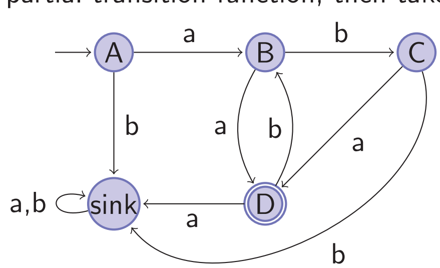


Training

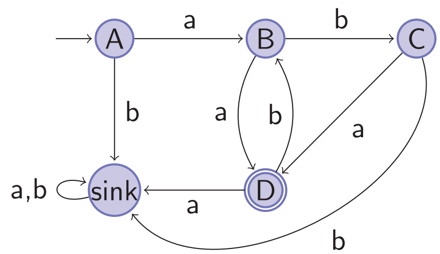
- Minimize the DFA



- The DFA has partial transition function, then take

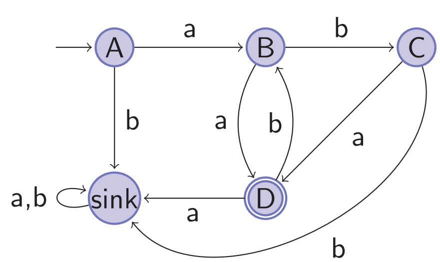


Training



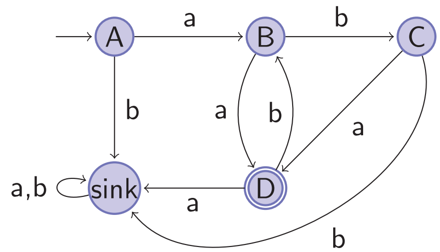
- Initial partition:
  $B_1 = \{D\};\ B_2 = \{A, B, C, sink\}$
- Split B_2 by $\mathrm{move}_d(A, a) \in B_2$ and $\mathrm{move}_d(B, a) \notin B_2$
- First refinement:
  $B_1 = \{D\};\ B_{21} = \{A, sink\};\ B_{22} = \{B, C\}$

Training



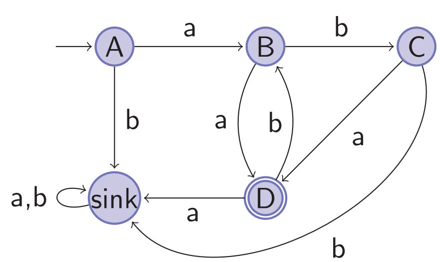
- First refinement:
  $B_1 = \{D\};\ B_{21} = \{A, sink\};\ B_{22} = \{B, C\}$
- Split B_{22} by $\mathrm{move}_d(B, b) \in B_{22}$ and $\mathrm{move}_d(C, b) \notin B_{22}$
- Second refinement:
  $B_1 = \{D\};\ B_{21} = \{A, sink\};\ B_{221} = \{B\}\ B_{222} = \{C\}$

Training



- Second refinement:
  $B_1 = \{D\}$; $B_{21} = \{A, sink\}$; $B_{221} = \{B\}$ $B_{222} = \{C\}$
- Split B_{21} by $\mathrm{move}_d(A, a) \in B_{221}$ and $\mathrm{move}_d(sink, a) \notin B_{221}$
- Last refinement:
  $B_1 = \{D\}$; $B_{211} = \{A\}$; $B_{212} = \{sink\}$; $B_{221} = \{B\}$ $B_{222} = \{C\}$

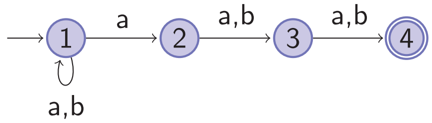
Training



- Last refinement:
  $B_1 = \{D\}$; $B_{211} = \{A\}$; $B_{212} = \{sink\}$; $B_{221} = \{B\}$ $B_{222} = \{C\}$
- Remove the sink and all the transitions to/from it, get the original DFA
- The original DFA was already minimal

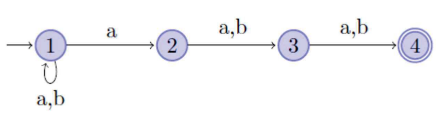
Training

- Find the minimal equivalent DFA



- First find the equivalent DFA
- Then minimize the DFA

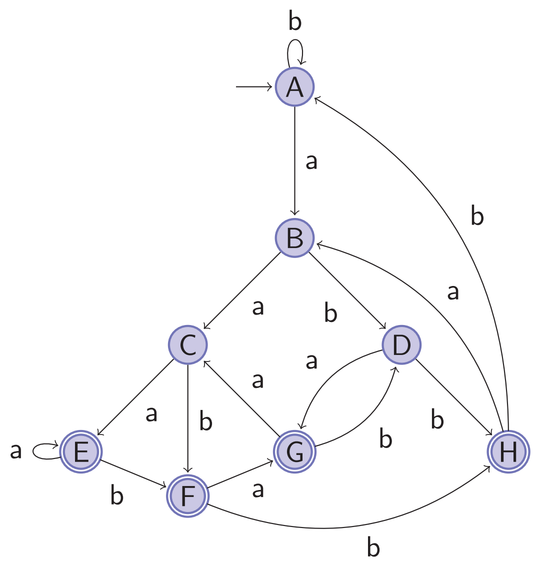
Training



		a	b
A={1}	initial	B	A
B={1,2}		C	D
C={1,2,3}		E	F
D={1,3}		G	H
E={1,2,3,4}	final	E	F
F={1,3,4}	final	G	H
G={1,2,4}	final	C	D
H={1,4}	final	B	A

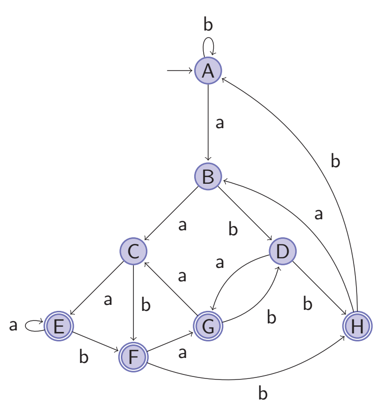
Training

DFA resulting from the subset construction



Training

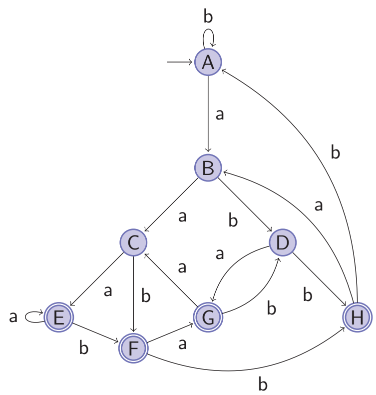
MINIMIZATION



- Partition: $\{A, B, C, D\}$; $\{E, F, G, H\}$
- Split: $\text{move}_d(A, a)$, $\text{move}_d(C, a)$
- Partition: $\{A, B\}$; $\{C, D\}$; $\{E, F, G, H\}$
- Split: $\text{move}_d(A, a)$, $\text{move}_d(B, a)$
- Partition: $\{A\}$; $\{B\}$; $\{C, D\}$; $\{E, F, G, H\}$
- Split: $\text{move}_d(G, b)$, $\text{move}_d(H, b)$
- Partition: $\{A\}$; $\{B\}$; $\{C, D\}$; $\{E, F, H\}$; $\{G\}$

Training

MINIMIZATION



- Partition: $\{A\}$; $\{B\}$; $\{C, D\}$; $\{E, F, H\}$; $\{G\}$
- Split: by $\mathrm{move}_d(C, a)$, $\mathrm{move}_d(D, a)$
- Partition: $\{A\}$; $\{B\}$; $\{C\}$; $\{D\}$; $\{E, F, H\}$; $\{G\}$
- Split: $\mathrm{move}_d(E, a)$, $\mathrm{move}_d(H, a)$
- Partition: $\{A\}$; $\{B\}$; $\{C\}$; $\{D\}$; $\{E, F\}$; $\{H\}$; $\{G\}$
- Split: $\mathrm{move}_d(E, a)$, $\mathrm{move}_d(F, a)$
- Partition: $\{A\}$; $\{B\}$; $\{C\}$; $\{D\}$; $\{E\}$; $\{F\}$; $\{G\}$; $\{H\}$
- Already minimal

Number of states of DFAs: worst case

LEMMA

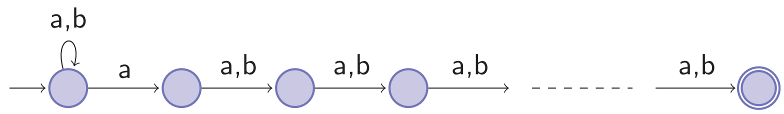
For each $n \in \mathbb{N}^+$ there is an NFA with $(n+1)$ states whose minimal equivalent DFA has at least 2^n states and total transition function

Number of states of DFAs: worst case

PROOF

- Take $\mathcal{L} = \mathcal{L}((a \mid b)^* a(a \mid b)^{n-1})$

- There is an NFA accepting $\mathcal{L}$ which has exactly $n + 1$ states



- Suppose, by contradiction, that a minimal DFA $\mathcal{D}$ exists which accepts $\mathcal{L}$ and has $k < 2^n$ states

- There are exactly 2^n distinct words over $\{a, b\}$ whose lenght is n

- Then there are two paths in $\mathcal{D}$
  - Whose lenght is n
  - Spell, respectively, w_1 and w_2 with $w_1 \neq w_2$
  - Share at least one node

Number of states of DFAs: worst case

PROOF

- Then, for some x_1, x_2, x, either
  $w_1 = x_1 ax$ *and* $w_2 = x_2 bx$ *or*
  $w_1 = x_1 bx$ *and* $w_2 = x_2 ax$

- Wlog suppose that $w_1 = x_1 ax$ and $w_2 = x_2 bx$

- Then $w_1' = x_1 ab^{n-1} \in \mathcal{L}(\mathcal{D})$

- Then the state reached by w_1' in $\mathcal{D}$ is final

- Contradiction: that state cannot be final because it is also reached by $x_2 bb^{n-1} \notin \mathcal{L}(\mathcal{D})$

Pumping Lemma for regular languages

LEMMA

Let $\mathcal{L}$ be a regular language. Then

- $\exists p \in \mathbb{N}^+$ such that
- $\forall z \in \mathcal{L}$ such that $|z| > p$
- $\exists u, v, w$ such that
  - $z = uvw$ and
  - $|uv| \leq p$ and
  - $|v| > 0$ and
  - $\forall i \in \mathbb{N}. uv^i w \in \mathcal{L}$

Pumping Lemma for regular languages

PROOF

- Let $\mathcal{L}$ be a regular language
- Then there exists a DFA $\mathcal{D} = (S, \mathcal{A}, \mathrm{move}_d, s_0, F)$ such that $\mathcal{L} = \mathcal{L}(\mathcal{D})$
- Let $p = |S| - 1$
- Then all the paths from s_0 to some final state that traverse every state at most once have their lengths bounded by p
- Then, by $|z| > p$, for some $a_1, \ldots, a_p$ and for some z', $z = a_1 \cdots a_p z'$ and at least one state, say s^*, is traversed more than once along the path $a_1 \cdots a_p$

Pumping Lemma for regular languages
PROOF

- Then there is a cycle in $\mathcal{D}$ that goes from s^* to s^* and that spells $a_{i+1} \cdots a_j$ for some i and j such that $i < j \leq p$
- Then let
  - $u = a_1 \cdots a_i$
  - $v = a_{i+1} \cdots a_j$
  - $w = \begin{cases} z' & \text{if } j = p \\ a_{j+1} \cdots a_p z' & \text{if } j < p \end{cases}$
- Then
  - $|uv| \leq p$
  - The length of v is at least 1
  - $uv^i w$ is accepted by $\mathcal{D}$ for every $i \in \mathbb{N}$

Pumping Lemma for regular languages
WHAT FOR

- Show by contradiction that a language is not regular
  - Assume the language be regular
  - Show that not(thesis) is true

- Thesis:
  $\exists p \in \mathbb{N}^+.\ \forall z \in \mathcal{L}\colon\ |z| > p.\ \exists u, v, w.\ P$
  where
  $P \equiv (z = uvw \text{ and } |uv| \leq p \text{ and } |v| > 0 \text{ and } \forall i \in \mathbb{N}. uv^i w \in \mathcal{L})$

- not(thesis):
  $\forall p \in \mathbb{N}^+.\ \exists z \in \mathcal{L}\colon\ |z| > p.\ \forall u, v, w.\ Q$
  where
  $Q \equiv (z = uvw \text{ and } |uv| \leq p \text{ and } |v| > 0)$ implies
  $(\exists i \in \mathbb{N}. uv^i w \notin \mathcal{L})$

Pumping Lemma at work

LEMMA

$\mathcal{L} = \{a^n b^n \mid n > 0\}$ is not regular

Pumping Lemma at work

PROOF

- Suppose $\mathcal{L}$ is regular, and let p be an arbitrary positive integer
- Take $z = a^p b^p$
- Observe that $\forall u, v, w$ if $(z = uvw$ and $|uv| \leq p$ and $|v| > 0)$
  - Then v contains only as, by $|uv| \leq p$
  - And v contains at least one a, by $|v| > 0$
- Then, for some $j > 0$, $uv^2w = a^p a^j b^p$
- Then $uv^2w \notin \mathcal{L}$ which contradicts the Pumping Lemma for regular languages

Training

- Let $\mathcal{L}_1$ be the language of words over $\{a, b\}$ with an odd number of occurrences of bs. Is $\mathcal{L}_1$ regular?
- Let $\mathcal{L}_2$ be the language of words over $\{a, b\}$ with an even number of occurrences of as. Is $\mathcal{L}_2$ regular?
- Let $\mathcal{L}_3$ be the language of words over $\{a, b\}$ with an even number of occurrences of as and an odd number of occurrences of bs. Is $\mathcal{L}_3$ regular?

Closure properties of regular languages

LEMMA
Regular languages are closed wrt

- union
- concatenation
- complementation
- intersection

Lexical analysis

Provide input to syntax analysis

Example grammar: c99_grammar.y

Typical choice of tokens:

- One token for each keyword
- Tokens for operators (or for classes of operators)
- One token for identifiers
- Tokens for punctuation symbols

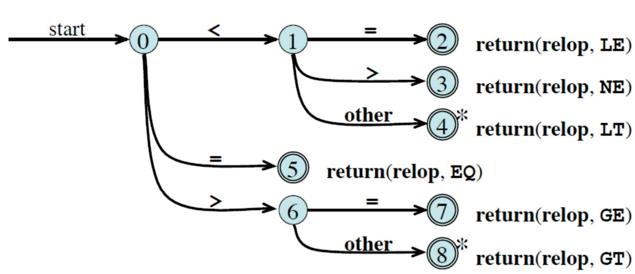
Task:

- Recognize lexemes
- Return tokens (pairs of token-name and token-value)

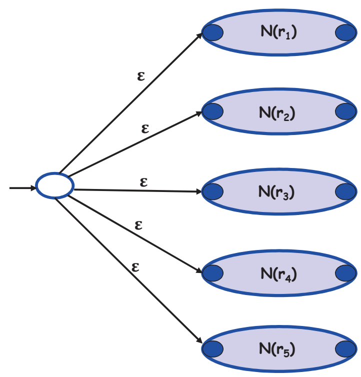
Lexical analysis

Lexemes

- Described by regular expressions
- Recognized by a state machine that can take appropriate actions when recognizing words



Pattern matching based on NFAs



Pattern matching based on NFAs

- Simulate the NFA
- Actions are associated with final states
- Find the longest match: continue simulating until no further move is possible
- If the reached set of states has associated actions, execute the "first" action
- Otherwise
  - Look backwards to the sequence of states
  - Pick up the first set of states containing at least a final one
  - Execute the "first" action
  - Update the pointer to the input buffer accordingly

Pattern matching based on DFAs obtained by subset construction is analogous

Lexical analysis generators

FLEX

- flex is normally distributed with C, and available from `http://flex.sourceforge.net/`

- Write a file `file.l` for flex
- Compile `file.l` with flex and get `lex.yy.c`
- Compile `lex.yy.c` with gcc and get the lexer

```
$> flex file.l
$> gcc lex.yy.c -lfl
$> ./a
```


Lexical analysis generators

FILES FOR FLEX

```
.....
%%
.....
%%
.....
```

```
%{ code
}%
shorthands for patterns
%%
pattern-1 {action-1};
pattern-2 {action-2};
....
%%
user routines, copied into lex.yy.c
```

Lexical analysis generators

DISAMBIGUATION

```
.....
%%
pattern-1 {action-1};
pattern-2 {action-2};
....
%%
.....
```

- Always take the longest match
- If there are more longest matches, take the first in the list

Lexical analysis generators

PATTERNS FOR FLEX

- Metacharacters

  `\ / - * + > '' { } . $ ( ) | % [ ] ^`

- Metacharacter matches

`.`	any character except newline
`\n`	newline
`*`	zero or more copies of
`+`	one or more copies of
`?`	zero or one copy of
`[]`	character class
`^`	beginning of line, negation if used in `[]`
`$`	end of line
`a|b`	`a` or `b`
`( )`	grouping
`"+"`	literal `"+"`
`{}`	regexp defined in the preamble