Lexical Anaylsis

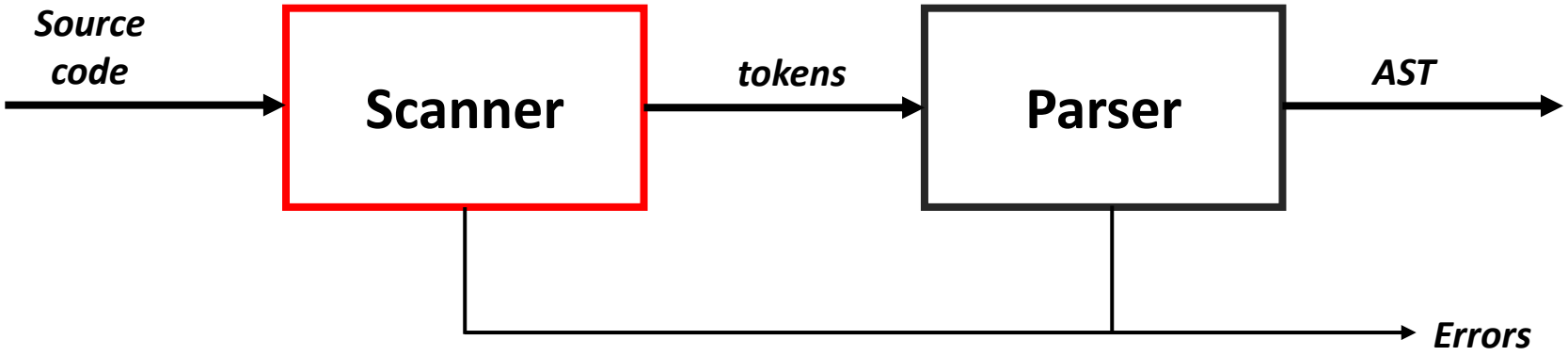
Jeonggeun Kim
Kyungpook National University

This lecture's materials are based on Prof. Bernd Burgstaller's slides (CSI4104 Compiler Design, Yonsei University)

Outline

- The role of a scanner
- Scanner concepts
  - Tokens, Lexemes, Patterns
- Regular Expression & Automata
  - Definitions of REs, DFAs and NFAs
  - REs → NFA
    - (Thompson's construction, Algorithm 3.3., Red Dragon book, Algorithm 3.23, Purple Dragon book)
  - NFA → DFA
    - (subset construction, Algorithm 3.2, Red Dragon book, Algorithm 3.20, Purple Dragon book)
  - DFA → minimal-state DFA
    - (state minimization, Algorithm 3.6, Red Dragon book, Algorithm 3.39, Purple Dragon book)
- Scanner generators

The Role of the Scanner



Scanner

- Maps stream of characters into words called **tokens**.
  - Basic units of syntax
  - x = y + 21; becomes…
    <id, x> <=> <id, y> <+> <intliteral, 21> <;>
- Characters that form a word are its **lexeme**.
- Formally, a token is a tuple <Token_type, value>.
- Informally, we often say “token x” when we mean <id, x>
- Tokens are to programming languages what words are to natural languages.

The Role of the Scanner

- Lexeme: a sequence of characters in the source program that matches the pattern for a token
  - a lexeme is identified by the lexical analyzer as an **instance** of that token.
- Token: a pair consisting of a token name and an optional attribute value.
  - token name is an **abstract symbol** representing a kind of lexical unit, e.g., a particular keyword or sequence of input characters (e.g., IDENTIFIER, ASSIGN)

[Token]	[Informal Description]	[Sample Lexemes]
if	characters i, f	if
else	characters e, l, s, e	else
comparison	< or > or <= or >= or == or !=	<=, !=
id	letter followed by letters and digits	pi, score, D2
number	any numeric constant	3.14159, 0, 6.02e23
literal	anything but ", surrounded by "'s	"core dumped"

x = a + b * 2
Lexemes: {x, =, a, +, b, *, 2}
Tokens: { <id, 0>, <=>, <id, 1>, <+>, <id, 2>, <*>, <id, 3> }

Tokens

- The tokens of our MiniC language are classified into token types:
  - identifiers: `i, j, initial, position, …`
  - keywords: `if, for, while, int, float, bool, …`
  - operators: `+ - * / <= && …`
  - separators: `{ } ( ) [ ] ; ,`
  - literals
  - integer literals: `0, 1, 22, …`
  - float literals: `1.25 1. .01 1.2e2 …`
  - bool literals: `true, false`
  - string literals: “`hello\n`”, “`string literal`”, `...`
- The exact token set depends on the given programming language. Pascal and Ada use “:=” for assignment, C uses “=”.
- Natural languages also contain different kinds of tokens (words): verbs, nouns, articles, adjectives, … The exact token set depends on the natural language in question.

Tokens can be described by patterns

- Pattern: a rule describing the set of lexemes of a particular token type.

Token type	Pattern (informal)	Set of lexemes
INTLITERAL	a string of decimal digits	{0, 1, 324, …}
ID	a string of letters, digits and underscores, beginning with a letter or underscore	{sum, _sum, facto, KNU_2023, …}
+	the character “+”	{+}
if	the characters “i”, “f”	{if}

- The pattern is said to match each string in the set.
- We want a formal notation for patterns.
  → allows us to specify the tokens of programming languages.

Tokens can be described by patterns

- Pattern: a rule describing the set of lexemes (=strings) of a particular token type.

Pattern name	Pattern	Set of lexemes (strings)
BinaryDigit	0 | 1	{0, 1}
DecimalDigit	0|1|2|3|4|5|6|7|8|9	{0,1,2,3,4,5,6,7,8,9}
TwoBinaryDigits	(0|1) (0|1)	{00, 01, 10, 11}
TwoBinaryDigits	BinaryDigit BinaryDigit	{00, 01, 10, 11}
Bananana	b a (n a)*	{ba, bana, banana, bananana, …}
Register	r (0|1|2|3|4|5|6|7|8|9)	{r0, r1, r2, … r9}

- The set of lexemes (=strings) described by pattern X is called the language of X.
- We write L(X) to denote the language of X.

String Concatenation

- If x and y are strings, then xy is the string formed by appending y to x.
- Examples:

x	y	xy
key	board	keyboard
java	script	javascript

Set Operations (Review)

Operation	*Definition*
Union of L and M, written $L \cup M$,	$L \cup M = \{s \mid s \in L \text{ or } s \in M\}$
Concatenation of L and M, written LM,	$LM = \{st \mid s \in L \text{ and } t \in M\}$
Kleene star (or closure) of L, written L^*,	$L^* = \bigcup_{0 \le i \le \infty} L^i$ $L^0 = \{\epsilon\}, \quad L^i = \underbrace{LL \ldots L}_{i \text{ times concatenation}}$ ϵ denotes the empty string
Positive Kleene star (or positive closure) of L, written L^+,	$L^+ = \bigcup_{1 \le i \le \infty} L^i$

Examples: Operations on Languages

- L = {a,...z, A,...Z}
- D = {0,...,9}

Example	Language
$L \cup D$	letters and digits
L^3	all 3-letter strings, e.g., abc, xyz, …
LD	strings consisting of a letter followed by a digit, e.g., a1, b2, x1
L^*	all strings of letters, including the empty string ε
$L(L \cup D)^*$	all strings of letters and digits, starting with a letter, e.g., x, y, a1, aa, c999ccc
D^+	all strings of one or more digits

Regular Expressions (inductive definition)

- **Regular expressions (REs)** can be used to describe patterns.
  - A regular expression r is a pattern that describes a set of lexemes L(r).
  - Lexemes are made from characters of a finite set ∑ called alphabet.
    - Notation: we underline the characters from ∑.

Precedence is closure, then concatenation, then alternation

- Inductive definition of regular expressions over alphabet ∑ :
- **R1:** ε is a RE denoting the set {ε}
- **R2:** If a is in ∑, then a is a RE denoting {a}
- If *x* and *y* are REs denoting *L(x)* and *L(y)* then
  - **R3:** *x* | *y* is a RE denoting *L(x)* ∪ *L(y)* — called "alternation"
  - **R4:** *xy* is a RE denoting *L(x)L(y)* — called "concatenation"
  - **R5:** *x** is a RE denoting *L(x)** — called "repetition" or "closure"

Example REs

- $\sum = \{0, 1\}$

Rules R2-R5 are from the previous slide…

RE	Language L(RE)
1	$L(1) \overset{R2}{=} \{1\}$
$0 \mid 1$	$L(0 \mid 1) \overset{R3}{=} L(0) \cup L(1) \overset{R2}{=} \{0\} \cup \{1\} = \{0, 1\}$
1^*	$L(1^*) \overset{R5}{=} L(1)^* \overset{R2}{=} \{1\}^* = \{\varepsilon, 1, 11, 111, \ldots\}$
$1^* 1$	$L(1^* 1) \overset{R4}{=} L(1^*)L(1) = \ldots = \{1, 11, 111, \ldots\}$
$0 \mid 1^* 0$	the set containing 0 and all strings consisting of zero or more 1's followed by a 0: $\{0, 10, 110, \ldots\}$
$(111)^*$	the set of strings that contain zero or more sequences of 111: $\{\varepsilon, 111, 111111, 111111111 \ldots\}$

Example REs (cont.)

- $\sum = \{a, b, c\}$

RE	Language L(RE)
$(ab)^*$	zero or more concatenations of the string ab $\{\varepsilon, ab, abab, ababab, \ldots\}$
$(a \mid b)^*$	zero or more concatenations of $(a \mid b)$ $L((a \mid b)^*) = (L(a \mid b))^* = (L(a) \cup L(b))^* = \{a, b\}^* =$ $\{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, aaaa, \ldots\}$
$(a \mid b)^* c$	strings from above, each string concatenated with character c $\{c, ac, bc, aac, abc, bac, bbc, aaac, aabc, \ldots\}$

Precedence Rules

- Precedence rules are needed to disambiguate regular expressions.
  - Like with arithmetic: a + b * c = a + (b * c)
  - Use parenthesis if you want (a + b) * c

- Precedence of regular expression operators is closure, then concatenation, then alternation.

- Examples:

  ab* = a(b*) use parenthesis if you want (ab)*

  ab|c = (ab) | c use parenthesis if you want a(b|c)

Example Token Specifications using REs

- digit: 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
  - Example digits: 0, 1, 2, …, 9

Notation: digit and letter are defined as symbolic names for REs.

- letter: a | b | ... | z | A | B | ... | Z
  - Example letters: a, b, c, …, Z
- Simple 2-character identifiers, starting with a letter, followed by a letter or digit:

  simple_id: letter (letter | digit)
  - Example for simple_id: a1, A1, ab, YD

Notation: simple_id uses the symbolic names for digit and letter (like a shorthand notation).

- 2-character identifiers, each character can be either a letter or a digit:

  twochar_id: (letter | digit) (letter | digit)
  - Example for twochar_id: a1, A1, 1A, xY, 12, 00

- What's the problem with **identifiers** consisting of digits only?

Example Token Specifications (cont.)

- Identifiers with at least one character, the first character must be a letter, the subsequent characters can be letters or digits:
  - **id: letter (letter | digit)***
  - Example for id: KNU, K, Daegu, cse2023, NeWJeaNS
- Integer numbers: first character is a digit, followed by zero or more digits:
  - **integer: digit digit***
  - Examples: 0, 1, 2, 01, 101, 999, 2023
- Signed integer numbers: an integer that can optionally have "+" or "-" in front:
  - **signed_integer: (+ | - | ε) integer**
  - Examples: 0, 1, +1, -1, -1557, +1557

Outline

- The role of a scanner ✓
- Scanner concepts ✓
  - Tokens, Lexemes, Patterns
- Regular Expression & Automata (to be continued…)
  - Definitions of REs (✓), DFAs and NFAs

Writing Regular Expressions

- Write a regular expression for each of the following sets of tokens:
  - Ruby binary literals consisting of “0b” followed by the binary number. Examples: 0b001011, 0b01.
  - Ruby binary literals, with an optional underscore (“_”) between binary digits. Examples: 0b0_101, 0b11_01, but not 0b_1 or 0b1_ .
  - Ada identifiers: a letter followed by any number of letters, digits, and underlines. An identifier must not end in an underline or have two underlines in a row.
  - A floating-point number: one or more digits (whole-number part) followed by a decimal point (“.”) and one or more digits (fractional part). Example: 3.14, 0.1234567
  - Floating point number in scientific notation: same as above, but optionally followed y “e” or “E”, and a signed integer exponent. Example: 1.2e-2, 2.3E+34, 2.3E34.

Shorthand Notations

- One or more concatenations: $r^+ = rr^*$
  - Denotes the language $(L(r))^+$
  - Same precedence and associativity as *
- Zero or one instance: r? = ε | r
  - Denotes the language $L(r) \cup \{\varepsilon\}$
  - Written as (r)? to indicate grouping, e.g., (12)?
- Character classes:
  - [a—z A—Z]

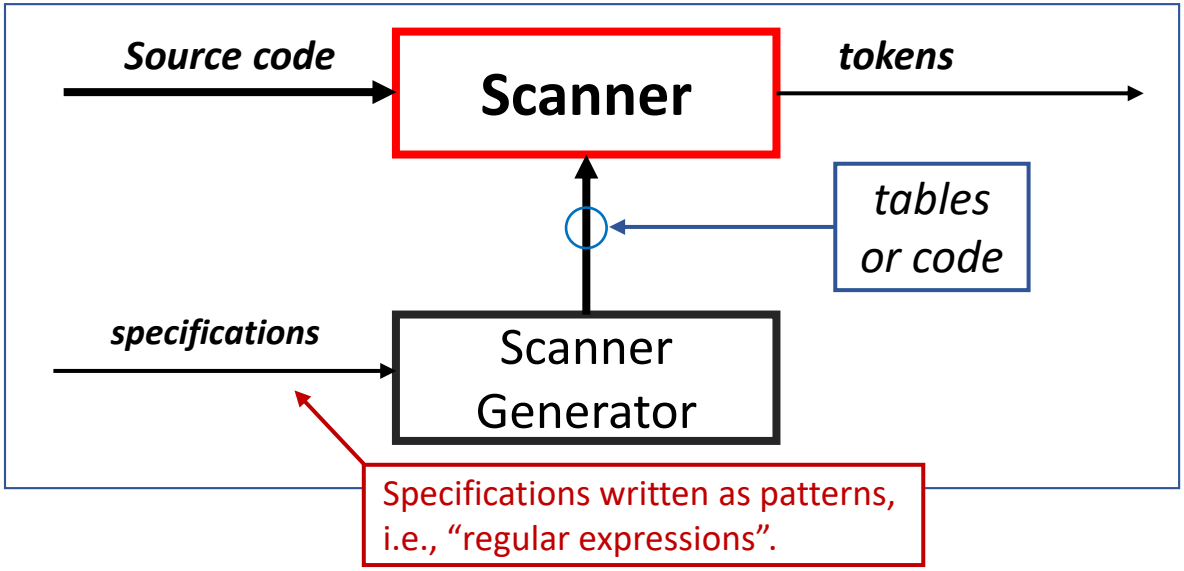
MiniC RE Examples

Token	RE
letter	[a—z A—Z_]
identifier	**letter (letter | digit)***
integer	**digit+**

- Note that with MiniC, letter includes the underscore character "_".
- In Java, letters and digits may be drawn from the entire Unicode character set.
  - Examples of Java identifier are abc リルパ 컴파

The Big Picture

- Why are we doing this?
  - We want to use regular expressions to specify scanners.
  - We want to harness the theory from classes like “Automata”.



- Goals:
  - To simplify specification & implementation of scanners
  - To understand the underlying techniques and technologies

Regular Expressions

- We use regular expressions to specify the mapping of lexemes to tokens.

- Using results from automata theory of algorithms, we can automatically build recognizers from regular expressions.

➔ We study REs and associated theory to automate scanner construction!

➔ Fortunately, the automatic techniques produce fast scanners. Used with text editors, URL filtering software, …

Example

- Consider the problem of recognizing register names

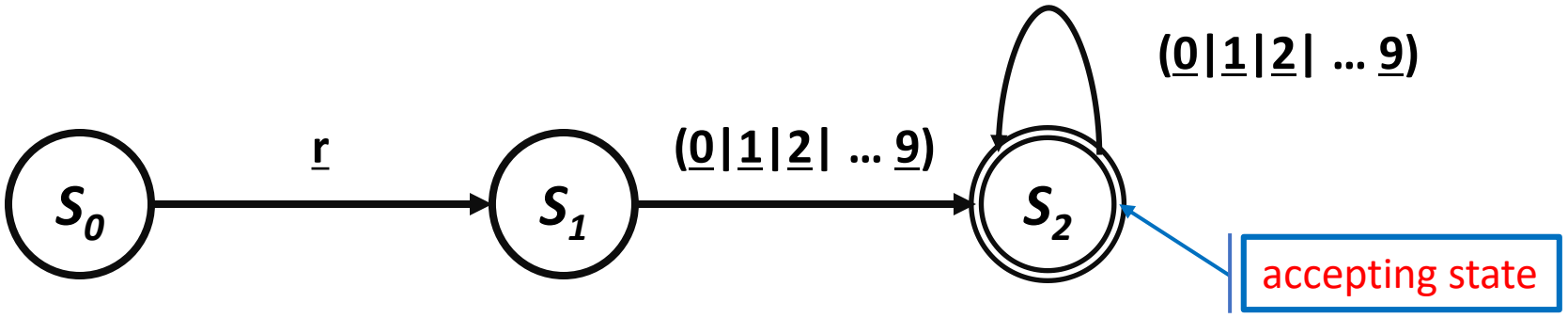
  Register: r (0|1|2| … |9) (0|1|2| … |9)*

  – Allows registers of arbitrary number

  – Requires at least one digit

  – ∑ = {r, 0, 1, …, 9}

- RE corresponds to a recognizer (or Deterministic Finite Automation, DFA)

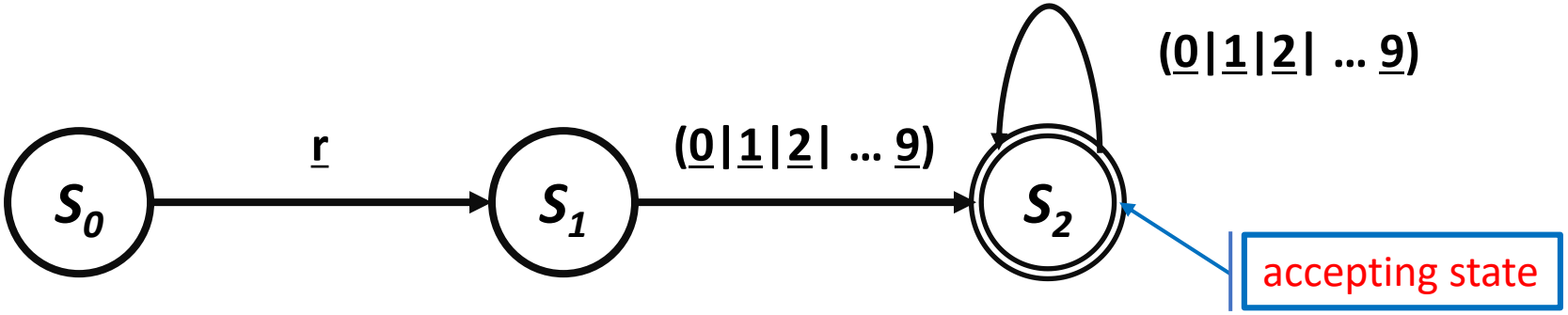


Recognizer for **Register (names)**

Transitions on other inputs go to an error state, s_e

Example (cont.)

- DFA operation
  - Starts in state S_0 and take transitions on each input character
  - DFA **accepts** a word *x* iff *x* leaves it in a final state (S_2)



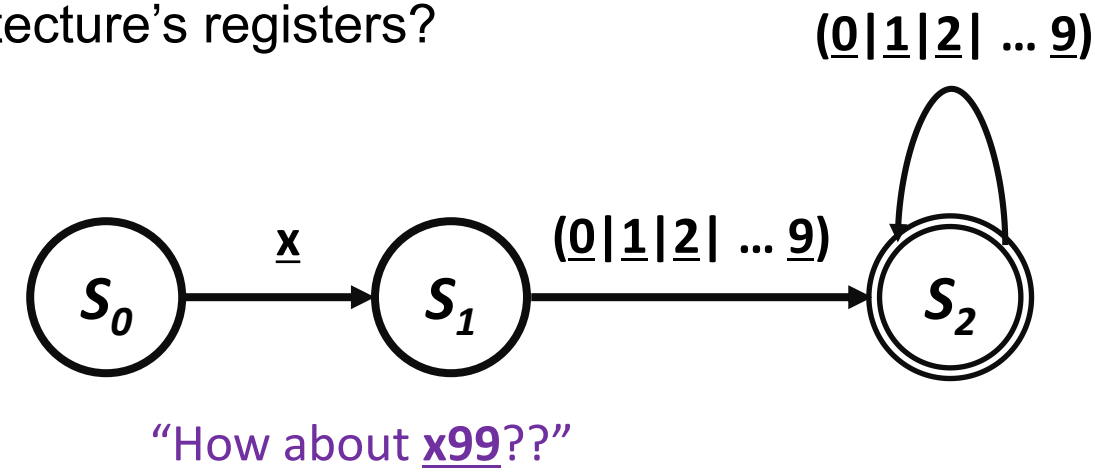
Recognizer for **Register**

- So,
  - r17 takes it through s_0, s_1, s_2 and accepts
  - r takes it through s_0, s_1 and fails
  - 1 takes it straight to s_e

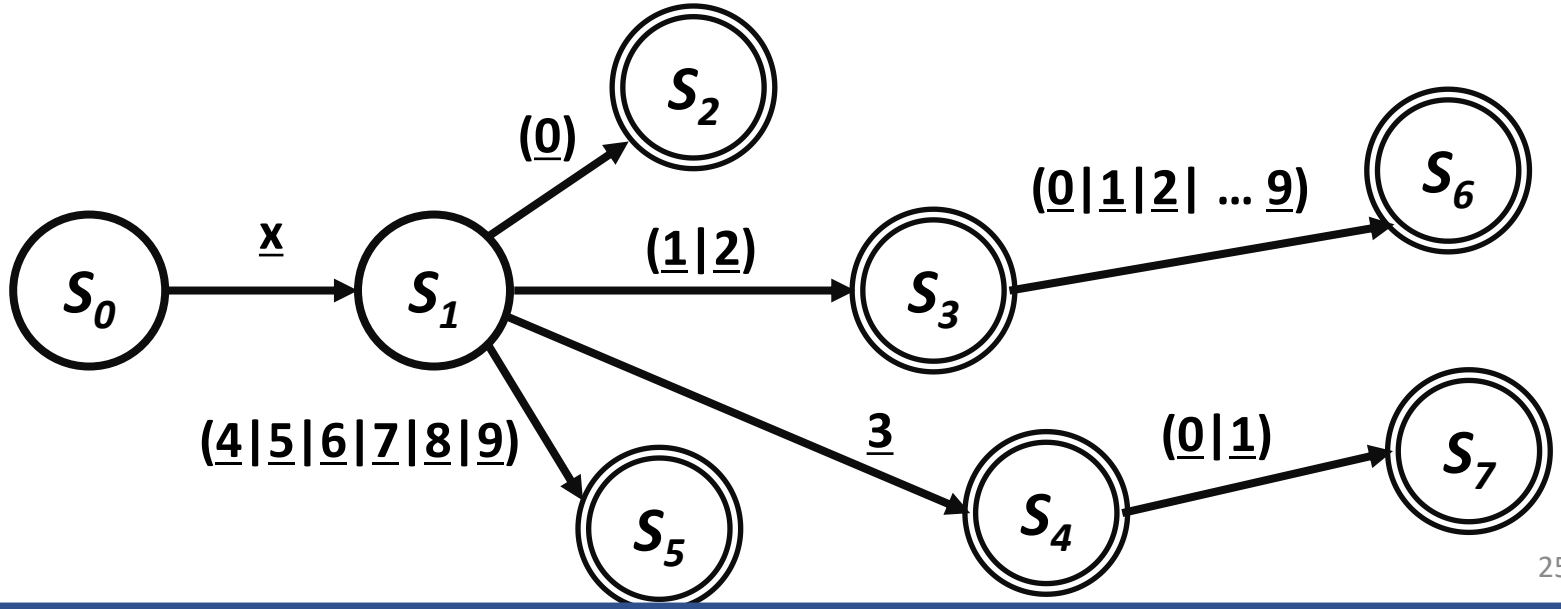
RISC-V Example

- What about RISC-V architecture's registers?

Register	ABI Name	Description
x0	zero	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary/alternate link register
x6–7	t1–2	Temporaries
x8	s0/fp	Saved register/frame pointer
x9	s1	Saved register
x10–11	a0–1	Function arguments/return values
x12–17	a2–7	Function arguments
x18–27	s2–11	Saved registers
x28–31	t3–6	Temporaries

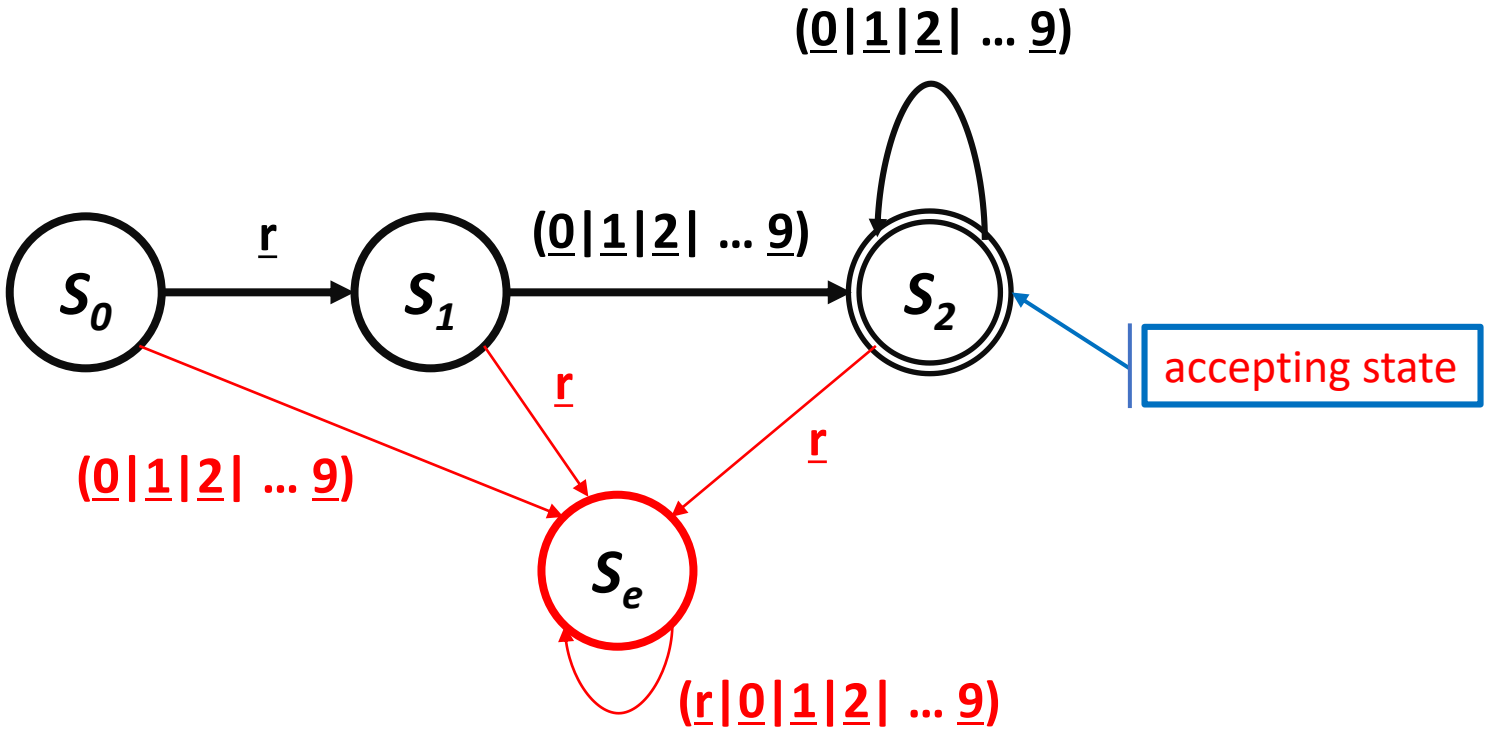


"How about **x99**??"



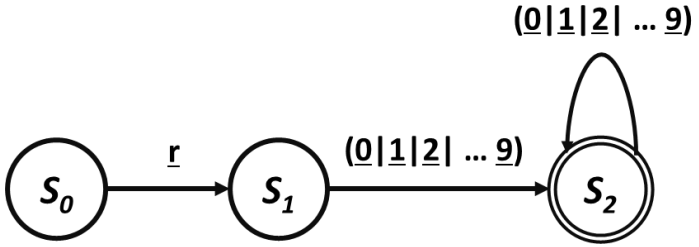
Error State

- Per convention, the error state, s_e, and transitions to it, are not drawn.
- Drawing the error state s_e, and transitions to it would give us:



Remember: ∑ = {r, 0, 1, …, 9}

Finite Automata (FA)



- A FA consists of a 5-tuple

$$< \sum, S, \delta, F, I >$$

- where
  - $\sum$ is an alphabet
  - S is a finite set of states
  - δ is a state transition function
  - F is a set of final or accepting states
  - I is the start state

Example:

$\sum = \{r, 0, 1, \ldots, 9\}$

$S = \{s_0, s_1, s_2\}$

δ = takes a state and a symbol as input and returns the next state (see next slide).

$F = \{s_2\}$

$I = \{s_0\}$

State Transition Function and Code

- To be useful, recognizer must turn into code

δ	r	0,1,2,3,4,5,6,7,8,9	All others
s_0	s_1	s_e	s_e
s_1	s_e	s_2	s_e
s_2	s_e	s_2	s_e
s_e	s_e	s_e	s_e

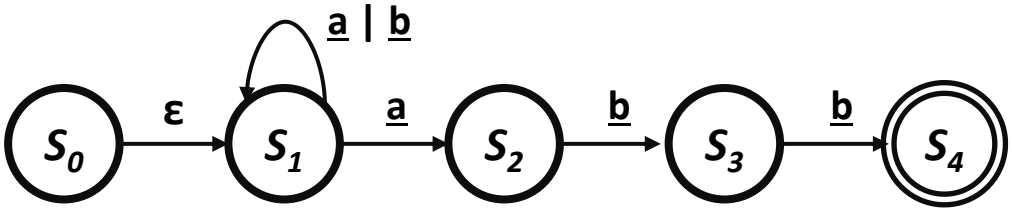
Table encoding RE

```
Char ← next character
State ← S0

while (Char ≠ EOF)
        State ← δ(State, Char)
        Char ← next character

if (State is a final state)
        then  report success
        else  report failure
```

Skeleton recognizer

Non-deterministic Finite Automata (NFAs)

- We can construct an FA for each RE, but…
- What about an RE such as **(a | b)* abb** ?



Problem: state s1 gives us two choices for character a: staying in s1 or going to s2. We have to guess the correct transition if we want to reach the accepting state s4.

* E.g., for string "abb": s0→s1→s2→s3→s4 (accept).

Guessing wrongly causes us to reject a valid string.

* E.g., for string "abb": s0→s1→s1→s1→s1 (fail).

- This is a little different than the FA before:
  - S_0 has a transition on **ε**
    - **ε** does not consume any input!
  - S_1 has **two** transitions on a

- This is a non-deterministic finite automaton (NFA)

Non-deterministic Finite Automata

- An NFA accepts a string x iff ∃ a path though the transition graph from s_0 to a final state such that the edge labels spell x
- Transitions on ε consume no input
- To "run" the NFA, start in s_0 and ***guess*** the right transition at each step (if there is more than one transition for a given symbol)
  - Always guess correctly
  - If some sequence of correct guesses accepts x then accept

- Why study NFAs?
  - They are the key to automating the RE → DFA construction
  - We can glue together NFAs with ε-transitions

Relationship between NFAs and DFAs

- DFA is a special case of an NFA
  - DFA has no ε transitions
  - DFA's transition function is single-valued
    - not possible to have 2 transitions from state s on a symbol a
  - Same rules will work

- DFA can be simulated with an NFA
  - ***Obviously***

- NFA can be simulated with a DFA *(less obvious)*
  - Simulate sets of possible states
  - Possible exponential blowup in the state space
  - Still, one state per character in the input stream

Summary: NFAs and DFAs

- A FA is a DFA if
  - no state has an **ε-transition**, i.e., there is no transition on input ε
  - for each state s and input symbol a, there is at most one edge labeled a leaving state s.

- A FA is an NFA
  - if it contains ε-transitions or
  - if it has several possible transitions from a state s on a given input symbol a.

Automating Scanner Construction

- To convert a specification to code:
  - 1) Write down the RE for the input language
  - 2) Build a big NFA
  - 3) Build the DFA that simulates the NFA
  - 4) Minimize the number of states in the DFA
  - 5) Generate the scanner code

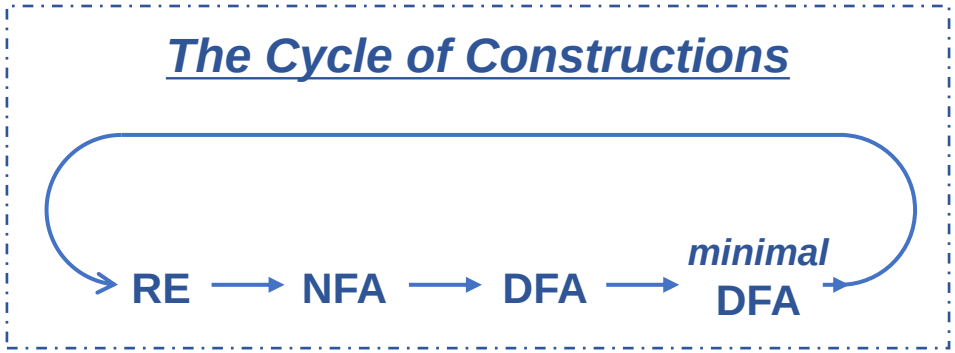
- Scanner generators
  - Lex and Flex work along these lines
  - Algorithms are well-known and well-understood
  - Key issue is interface to parser
  - You could build one in a weekend!

Outline

- The role of a scanner ✓
- Scanner concepts ✓
  - Tokens, Lexemes, Patterns
- Regular Expression & Automata (to be continued…)
  - Definitions of REs, DFAs and NFAs ✓
  - REs→NFA
    (Thompson's construction, Algorithm 3.3, Red Dragon book, Algorithm 3.23, Purple Dragon book)
  - NFA→DFA (subset construction, Algorithm 3.2, Red Dragon book, Algorithm 3.20, Purple Dragon)
  - DFA→minimal-state DFA
    (state minimization, Algorithm 3.6, Red Dragon book, Algorithm 3.39, Purple Dragon book)
- Scanner generators

Automating Scanner Construction

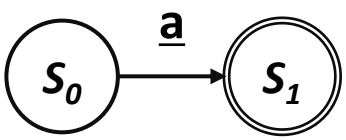
- RE → NFA *(Thompson's construction)*
  - Build an NFA for ε and each symbol a ∈ ∑ occurring in the RE
  - Combine them with ε-moves
- NFA → DFA *(subset construction)*
  - Build the simulation of the NFA
- DFA → Minimal DFA
  - Hopcroft's algorithm



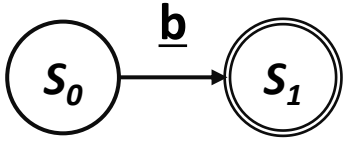
- DFA → RE *(Not part of the scanner construction)*
  - All pairs, all paths problem
  - Take the union of all paths from s_0 to an accepting state

RE → NFA using Thompson's Construction

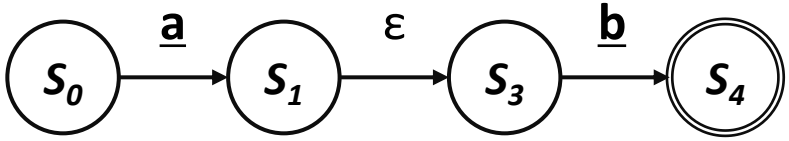
- Key idea:
  - construct NFA for each symbol and each operator of the RE
  - Join them with ε moves in precedence order of RE operators



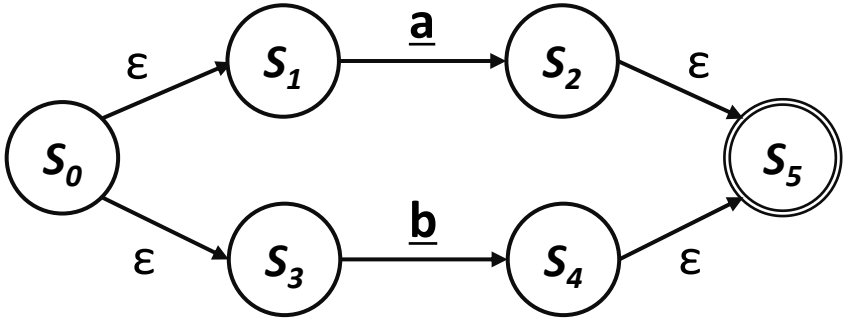
NFA for a



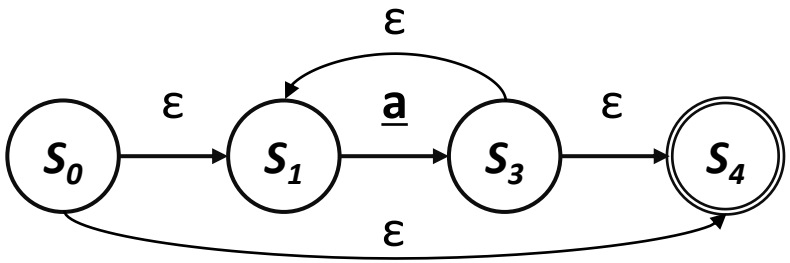
NFA for b



NFA for ab



NFA for a | b

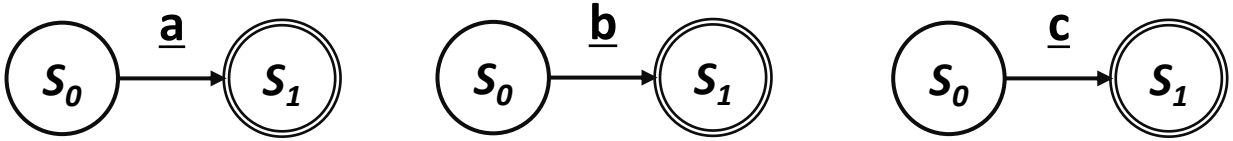


NFA for a^*

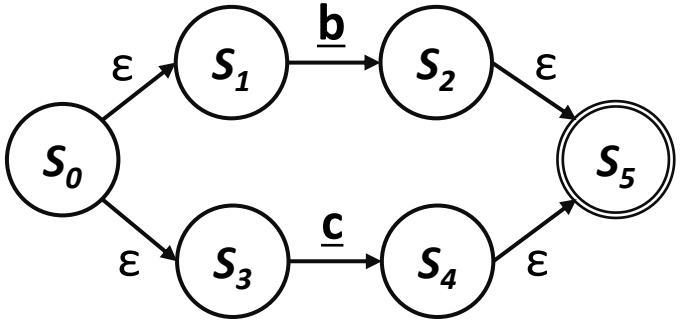
Note: when joining two NFAs, we renumber states so that s_0 is again the start state, and all state names are unique.

Example of Thompson's Construction

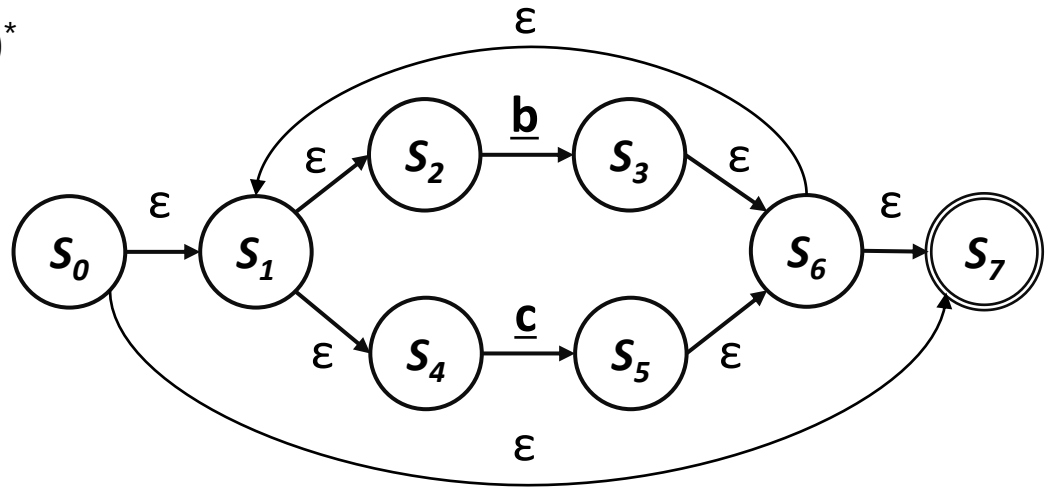
- Let's try **a (b | c)***
  - 1. a, b, and c:



  - 2. b | c

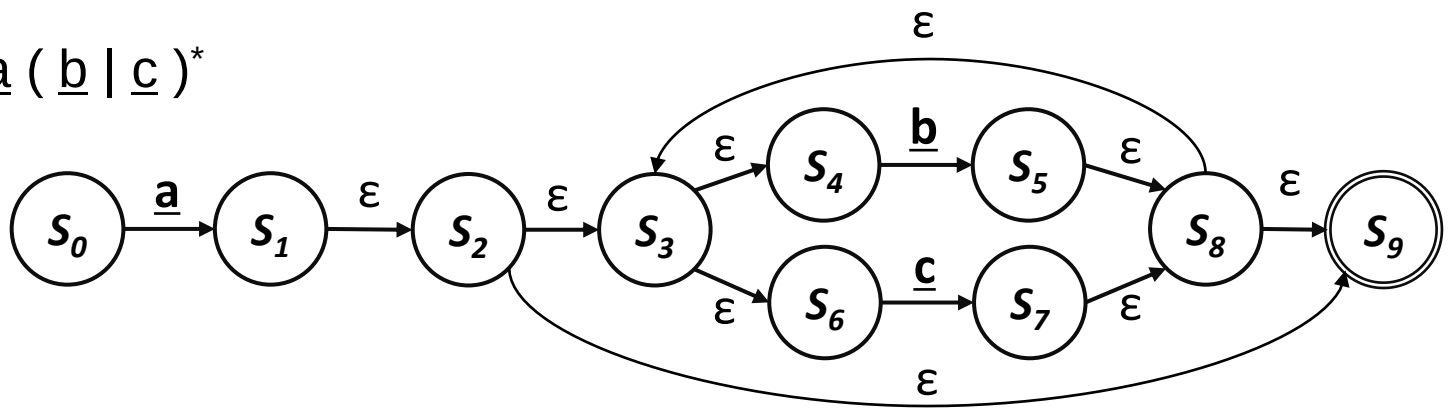


  - 3. (b | c)*

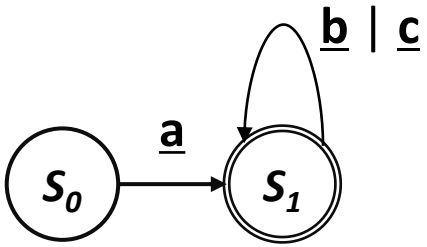


Example of Thompson's Construction (cont.)

- 4. a (b | c)*



Of course, a human would design something simpler …



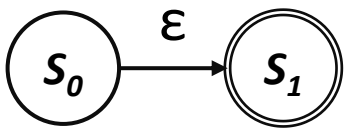
But, we can automate the production of the more complex automaton …

Thompson's Construction

- Syntax-driven
  - follows the structure of REs
- **Inductive:** the cases in the construction of the NFA follow the cases in the definition of REs.
- **Important:** if a symbol a occurs several times in a RE r, then a separate NFA is constructed for each occurrence of a.

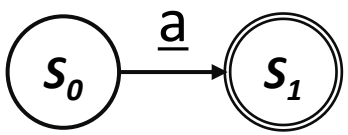
Thompson's Construction

- **Inductive Base:**
  - For ε, construct the NFA



Thompson's Construction for automata follows the inductive definition of regular expressions from page 68.

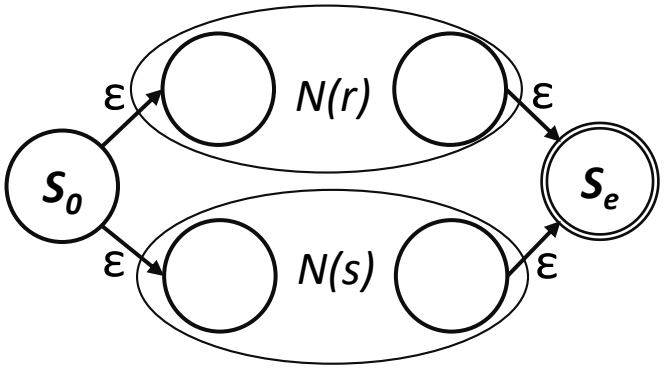
  - For $\underline{a} \in \sum$, construct the NFA

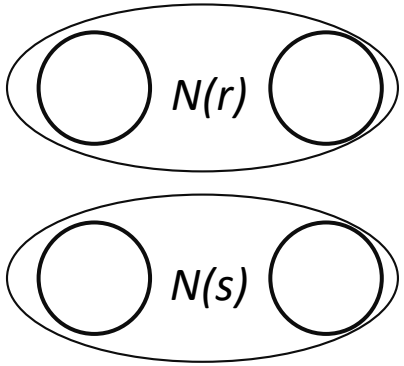


- **Inductive step:** suppose N(r) and N(s) are NFAs for REs r and s.
  - Then… (continued on next slide).

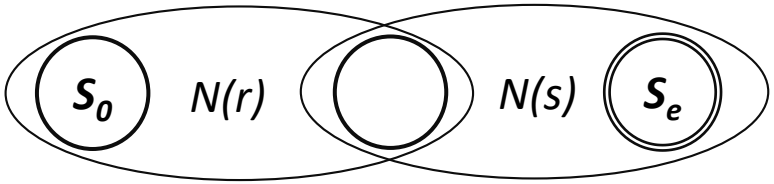
Thompson's Construction

- RE *r* | *s*:



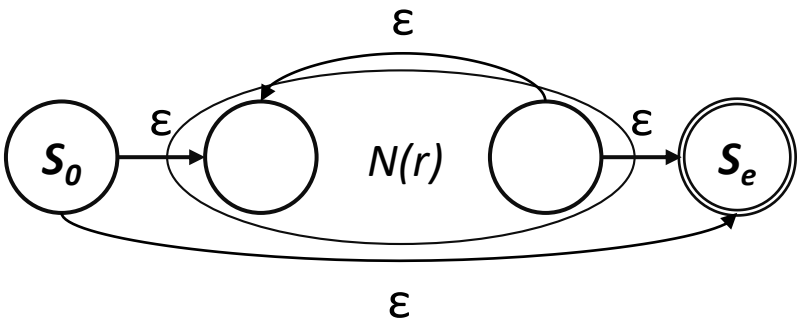


- RE *r s*:



Thompson's Construction for automata follows the inductive definition of regular expressions from page 68.

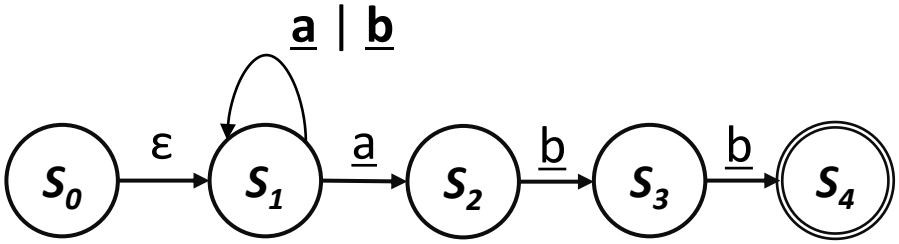
- RE r^*:



Outline

- The role of a scanner ✓
- Scanner concepts ✓
  - Tokens, Lexemes, Patterns
- Regular Expression & Automata (to be continued…)
  - Definitions of REs, DFAs and NFAs ✓
  - REs→NFA ✓
    (Thompson's construction, Algorithm 3.3, Red Dragon book, Algorithm 3.23, Purple Dragon book)
  - NFA→DFA (subset construction, Algorithm 3.2, Red Dragon book, Algorithm 3.20, Purple Dragon)
  - DFA→minimal-state DFA
    (state minimization, Algorithm 3.6, Red Dragon book, Algorithm 3.39, Purple Dragon book)
- Scanner generators

Subset Construction (NFA→DFA)



Instead of guessing in state s_1 on symbol a, we can follow both transitions ($s_1 \rightarrow s_1$ and $s_1 \rightarrow s_2$) in parallel.

- We introduce a "virtual state" that combines the states that we reach from s_1 on symbol a.
- The new virtual state contains states s_1 and s_2, we write $\{s_1, s_2\}$ for the virtual state.
- **A question:** if we are in the virtual state $\{s_1, s_2\}$, where can we go when we read symbol b?

Problem: state s_1 gives us two choices for character a: staying in s_1 or going to s_2. We have to guess the correct transition if we want to reach the accepting state s_4.

E.g., for string "abb": s0->s1->s2->s3->s4 (accept).

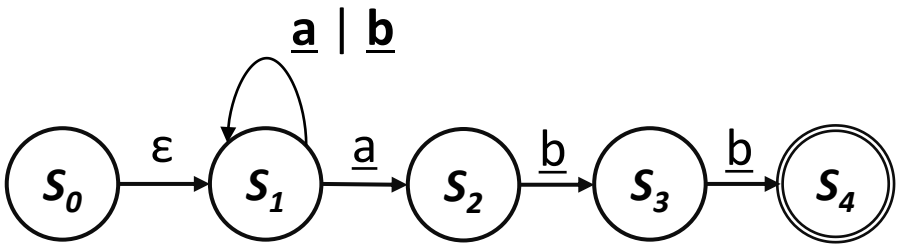
Guessing wrongly causes us to reject a valid string.

E.g., for string "abb": s0->s1->s1->s1->s1 (fail).

- **Answer:** the virtual state takes us anywhere that one of its member states takes us on symbol b (either s_1 or s_3 in the above NFA). So we introduce a second virtual state $\{s_1, s_3\}$.

(continued on next slide.)

Subset Construction (NFA→DFA)



- A "virtual state" is a subset of the set of states S={S_0, S_1, S_2, S_3, S_4} of the NFA.

δ	a	b
{s_0, s_1}	{s_1, s_2}	{s_1}
{s_1, s_2}	{s_1, s_2}	{s_1, s_3}
{s_1}	{s_1, s_2}	{s_1}
{s_1, s_3}	{s_1, s_2}	{s_1, s_4}
{s_1, s_4}	{s_1, s_2}	{s_1}

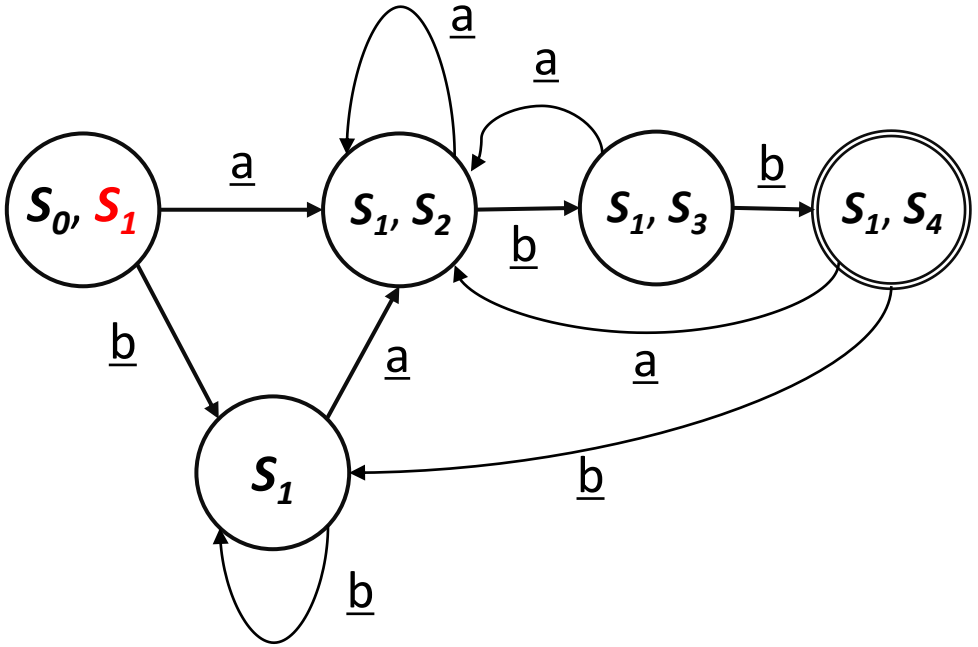


Table encoding DFA

Algorithm: NFA→DFA with Subset Construction

We need to build a simulation of the NFA

Two key functions

- Move(s_i , a) : gives set of states reachable from set s_i by a
- ε-closure(s_i) : gives set of states reachable from set s_i by ε

The algorithm:

- Start state derived from s_0 of the NFA:
  - Take its ε-closure S_0 = ε-closure({s_0})
- Take the image of S0, Move(S0, α), for each α∈∑, and take its ε-closure
- Iterate until no more states are added

Sounds more complex than it is…

Algorithm: NFA→DFA with Subset Construction

The algorithm:

```
Dstates ← { };
add ε-closure(s0) as an
      unmarked state to Dstates;
while ( there is an unmarked
         state T in Dstates ) {
      mark T;
      for each α ∈ ∑ {
            U ← ε-closure(Move(T, α))
            if ( U ∉ Dstates ) then
               add U as an unmarked
                        state to Dstates;
               δ [T, α] ← U
      }
}
```

Let's think about why this works

The algorithm halts:

1. **Dstates contains no duplicates (test before adding)**
2. **$2^{|S|}$ is finite**
3. **while loop adds to *Dstates*, but does not remove from *Dstates (monotone)***
   ➔ the loop halts

Dstates contains all the reachable NFA states:
It tries each character $\alpha \in \sum$ in each state T.

It builds every possible NFA configuration.
➔ *Dstates and δ form the DFA*

Any DFA state containing a final state from the NFA becomes a final state in the DFA.

NFA→DFA with Subset Construction

Example of a fixed-point computation

- Monotone construction of some finite set
- Halts when it stops adding to the set
- Proofs of halting & correctness are similar
- These computations arise in many contexts

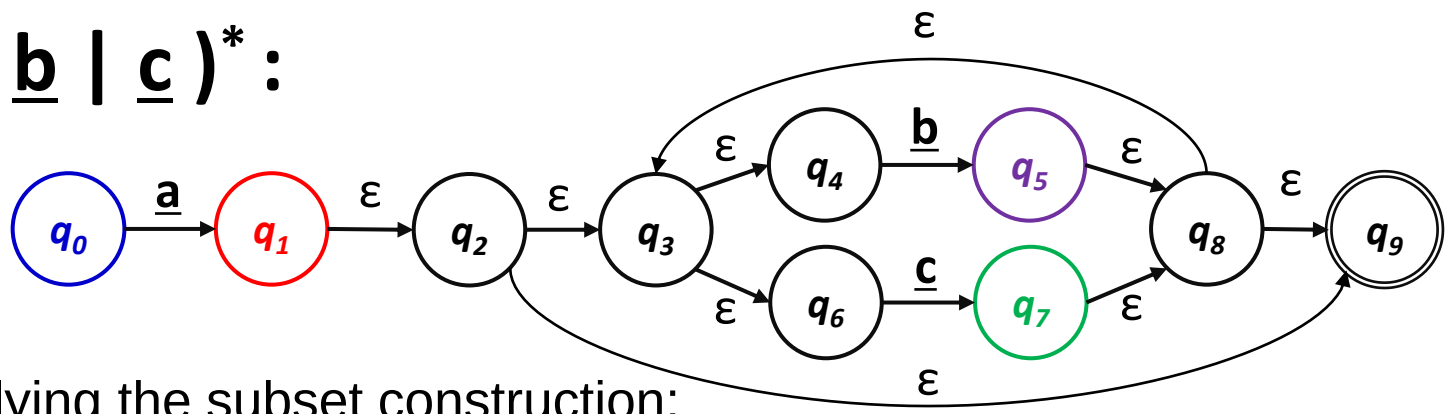
Other fixed-point computations

- Classic data-flow analysis (& Gaussian Elimination)
  - Solving sets of simultaneous set equations

We will see many more fixed-point computations

Example: NFA→DFA with Subset Construction

a (b | c)* :



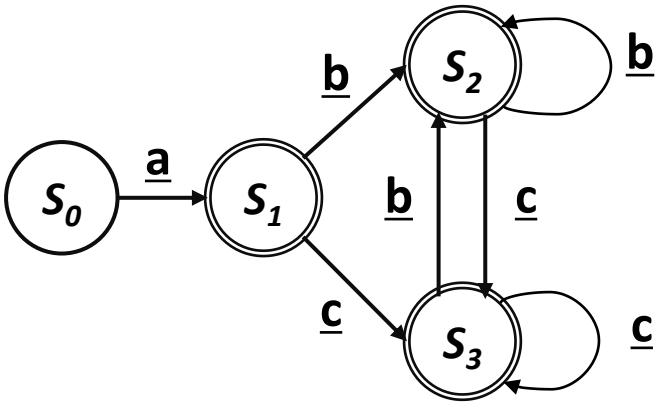
- Applying the subset construction:

	NFA states	ε-closure (Move (s, *))		
		a	**b**	**c**
S_0	q_0	$q_1, q_2, q_3, q_4, q_6, q_9$	none	none
S_1	$q_1, q_2, q_3, q_4, q_6, q_9$	none	$q_5, q_8, q_9, q_3, q_4, q_6$	$q_7, q_8, q_9, q_3, q_4, q_6$
S_2	$q_5, q_8, q_9, q_3, q_4, q_6$	none	S_2	S_3
S_3	$q_7, q_8, q_9, q_3, q_4, q_6$	none	S_2	S_3

Final states

Example: NFA→DFA with Subset Construction

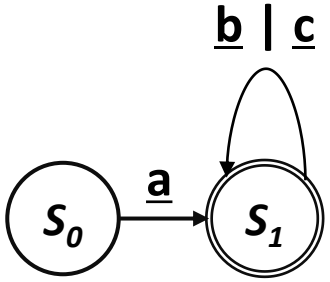
- The DFA for a (b | c)*



δ	a	b	c
S_0	S_1	-	-
S_1	-	S_2	S_3
S_2	-	S_2	S_3
S_3	-	S_2	S_3

- Ends up smaller than the NFA
- All transitions are deterministic
- Use the same code skeleton as before

But, remember our goal:



Outline

- The role of a scanner ✓
- Scanner concepts ✓
  - Tokens, Lexemes, Patterns
- Regular Expression & Automata (to be continued…)
  - Definitions of REs, DFAs and NFAs ✓
  - REs→NFA ✓
    (Thompson's construction, Algorithm 3.3, Red Dragon book, Algorithm 3.23, Purple Dragon book)
  - NFA→DFA ✓
    (subset construction, Algorithm 3.2, Red Dragon book, Algorithm 3.20, Purple Dragon)
  - DFA→minimal-state DFA
    (state minimization, Algorithm 3.6, Red Dragon book, Algorithm 3.39, Purple Dragon book)
- Scanner generators

DFA Minimization Overview

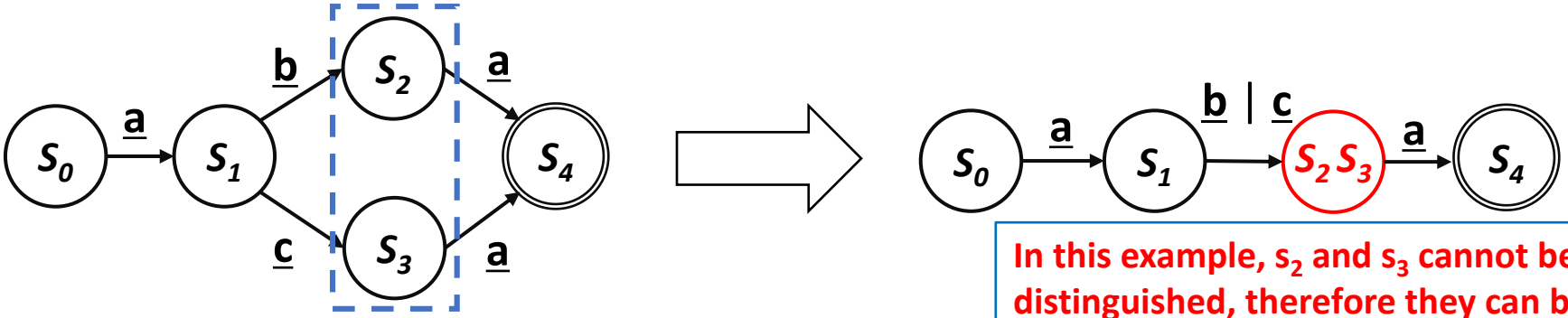
Two states p and q are ***indistinguishable*** by input string **w**
iff the DFA starting in p accepts w and starting in q accepts w, **and**
the DFA starting in p does not accept w and starting in q does not accept w.

The Big Picture:

- Discover distinguishable states
- States that cannot be distinguished can be represented by a single state.

Two states p and q are distinguishable by input **string w** iff the DFA starting in p accepts **w** but starting in q does not accept **w**.

Two states p and q are distinguishable if they are distinguishable by some input string w.



In this example, s_2 and s_3 cannot be distinguished, therefore they can be represented by a single state $s_2 s_3$.

DFA Minimization Overview

- The set of states is divided into subsets of states that cannot be distinguished.
- We say that the set of states S is partitioned into **partition** P:
  - Each state $s \in S$ is in exactly one set $p_i \in P$
  - States in the same set have not been distinguished yet.
  - States from different sets are known to be distinguishable.

Step 1:

- **Initially** the partition **P** consists of **2 sets**:
  - the accepting states **F** and the non-accepting states **S – F**.
  - States from F and S-F are distinguishable by ε.

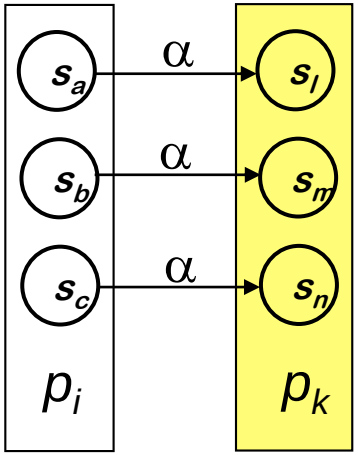
Step 2:

- The minimization algorithm **repeatedly picks a set $p_i \in P$** and tries to distinguish between its states by some symbol $\alpha \in \sum$.

  → split p_i along α (see next slide).

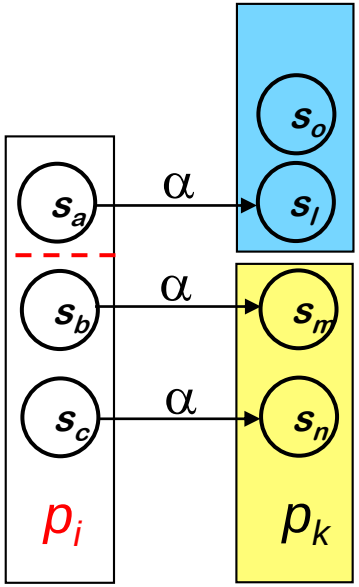
Splitting of a State Set along Symbol α

- The minimization algorithm repeatedly picks a set $p_i \in P$ and tries to distinguish between its states by some symbol $\alpha \in \sum$:



α does not split p_i

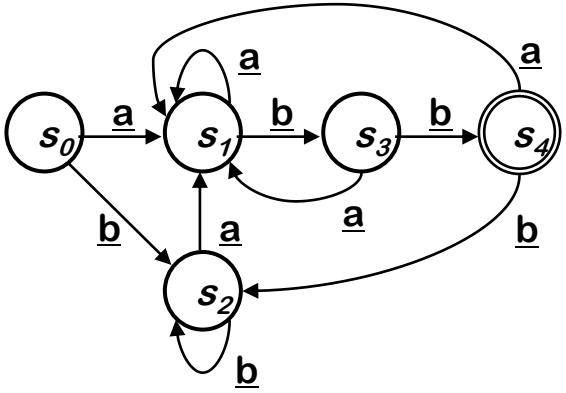
α does not split p_i



α splits p_i into $\{s_a\}$ and $\{s_b, s_c\}$

$\{s_a\}$ and $\{s_b, s_c\}$ are distinguishable by α…

- Eventually no more set can be split and the algorithm terminates.

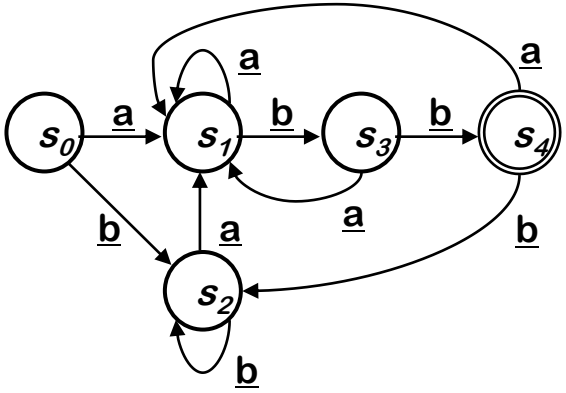


Minimization of DFA

$P_0 = \{ \{s_4\}, \{s_0, s_1, s_2, s_3\} \}$

Start off with 2 subsets of S: *F and S-F*
($p \in 2^S$ (powerset)**)**

```
P ← { F, S-F }
while ( P is still changing )
        T ← { }
        for each set p ∈ P
                T ← T ∪ Split (p)
        P ← T

Split (p)
        for each α ∈ ∑
        {
                if α splits p into p1 and p2
                then return {p1, p2}
        }
        return p;
```

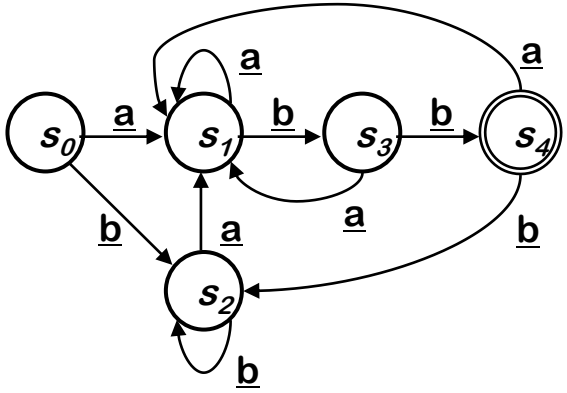


Minimization of DFA

$P_0 = \{ \{s_4\}, \{s_0, s_1, s_2, s_3\} \}$

$T = \{ \}$

```
P ← { F, S-F }
while ( P is still changing )
        T ← { }
        for each set p ∈ P
                T ← T ∪ Split (p)
        P ← T

Split (p)
        for each α ∈ ∑
        {
                if α splits p into p1 and p2
                then return {p1, p2}
        }
        return p;
```



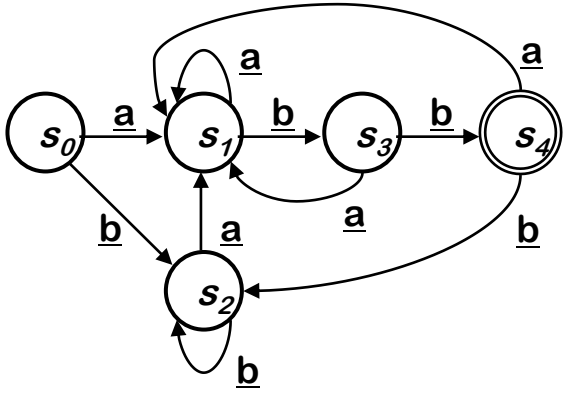
Minimization of DFA

$\mathbf{P_0}$ = { $\{s_4\}$, $\{s_0, s_1, s_2, s_3\}$ }

T = { }

p = $\{s_4\}$

```
P ← { F, S-F }
while ( P is still changing )
        T ← { }
        for each set p ∈ P
                T ← T ∪ Split (p)
        P ← T

Split (p)
        for each α ∈ ∑
        {
                if α splits p into p1 and p2
                then return {p1, p2}
        }
        return p;
```



Minimization of DFA

$P_0 = \{ \{s_4\}, \{s_0, s_1, s_2, s_3\} \}$

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

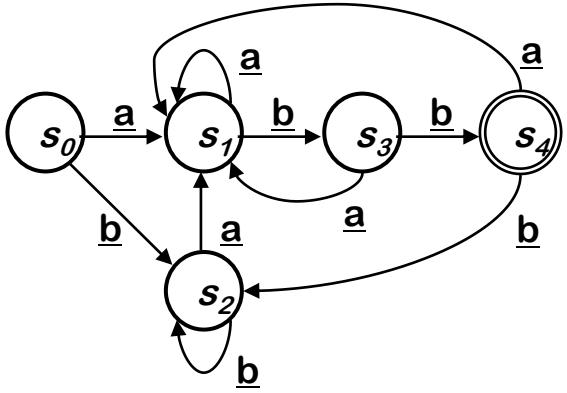
$T = \{ \}$

$p = \{s_4\}$

```
P ← { F, S-F }
while ( P is still changing )
        T ← { }
        for each set p ∈ P
                T ← T ∪ Split (p)
        P ← T

Split (p)
        for each α ∈ ∑
        {
                if α splits p into p1 and p2
                then return {p1, p2}
        }
        return p;
```

- Invoke Split(p) function:
  - Determines whether p can be split into p_1 and p_2
  - How?
    - If these two states can be distinguishable by a symbol α…

If X and Y are two states in a DFA,
we can combine these two states into {X, Y}
when they are not distinguishable.



Minimization of DFA

$\mathbf{P_0} = \{ \{s_4\}, \{s_0, s_1, s_2, s_3\} \}$

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

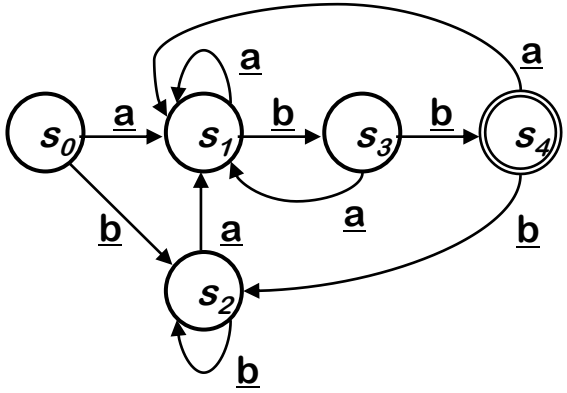
$\mathbf{T} = \{ \}$

$p = \{s_4\}$

$\alpha = a$

```
P ← { F, S-F }
while ( P is still changing )
        T ← { }
        for each set p ∈ P
                T ← T ∪ Split (p)
        P ← T

Split (p)
        for each α ∈ ∑
        {
                if α splits p into p1 and p2
                then return {p1, p2}
        }
        return p;
```

If X and Y are two states in a DFA,
we can combine these two states into {X, Y}
when they are not distinguishable.



Minimization of DFA

$P_0 = \{ \{s_4\}, \{s_0, s_1, s_2, s_3\} \}$

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

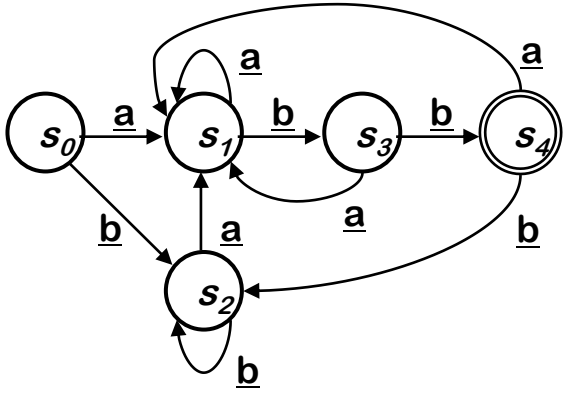
$T = \{\ \}$

$p = \{s_4\}$

α = a

```
P ← { F, S-F }
while ( P is still changing )
        T ← { }
        for each set p ∈ P
                T ← T ∪ Split (p)
        P ← T

Split (p)
        for each α ∈ ∑
        {
                if α splits p into p1 and p2
                then return {p1, p2}
        }
        return p;
```

- Does **a** can split p = $\{s_4\}$ to p_1 and p_2?
  - ⇔ Are there any two subsets can be distinguishable in set p by symbol **a**?
    - → Nope, not this time. (we cannot split a set which has a single element…)



Minimization of DFA

$P_0 = \{ \{s_4\}, \{s_0, s_1, s_2, s_3\} \}$

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

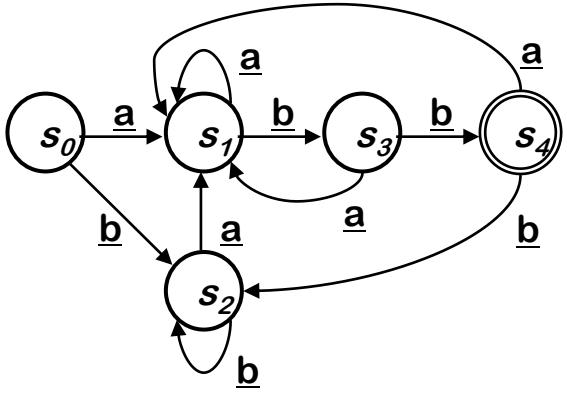
$T = \{ \}$

$p = \{s_4\}$

α = b

```
P ← { F, S-F }
while ( P is still changing )
        T ← { }
        for each set p ∈ P
                T ← T ∪ Split (p)
        P ← T

Split (p)
        for each α ∈ ∑
        {
                if α splits p into p1 and p2
                then return {p1, p2}
        }
        return p;
```

- Does **b** can split $p = \{s_4\}$ to p_1 and p_2?
  - ⇔ Are there any two subsets can be distinguishable in set p by symbol **b**?
    - → Nope.
      (we cannot split a set which has a single element…)



Minimization of DFA

$P_0 = \{ \{s_4\}, \{s_0, s_1, s_2, s_3\} \}$

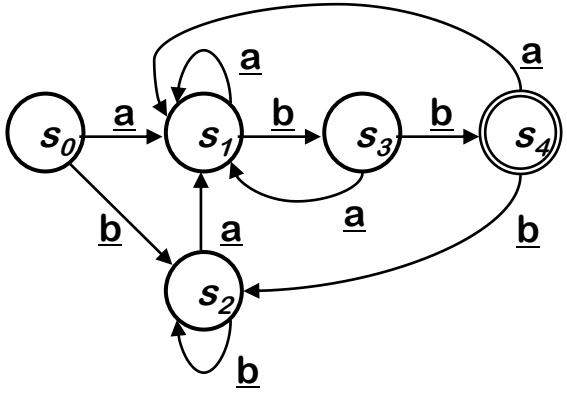
$\sum = \{a, b\}$

$T = \{ \}$

$p = \{s_4\}$

$\alpha =$

```
P ← { F, S-F }
while ( P is still changing )
        T ← { }
        for each set p ∈ P
                T ← T ∪ Split (p)
        P ← T

Split (p)
        for each α ∈ ∑
        {
                if α splits p into p1 and p2
                then return {p1, p2}
        }
        return p;
```



Minimization of DFA

$P_0 = \{ \{s_4\}, \{s_0, s_1, s_2, s_3\} \}$

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

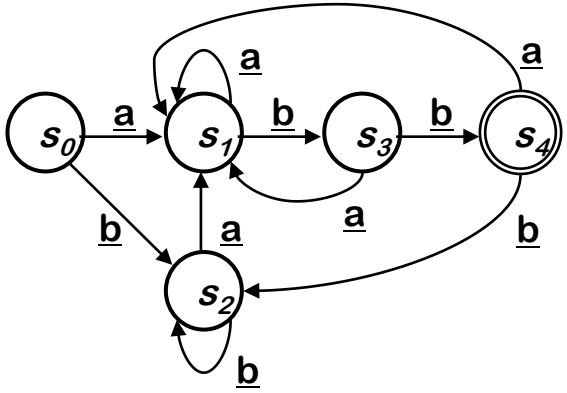
$T = \{ \}$

$p = \{s_4\}$

$\alpha =$

$T = \{ \} \cup \text{Split}(\{s_4\}) = \{s_4\}$

```
P ← { F, S-F }
while ( P is still changing )
        T ← { }
        for each set p ∈ P
                T ← T ∪ Split (p)
        P ← T

Split (p)
        for each α ∈ ∑
        {
                if α splits p into p1 and p2
                then return {p1, p2}
        }
        return p;
```



Minimization of DFA

$\mathbf{P_0} = \{ \{s_4\}, \{s_0, s_1, s_2, s_3\} \}$

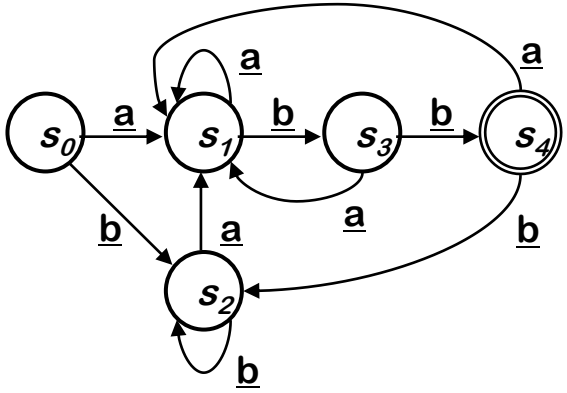
$\sum = \{a, b\}$

$\mathbf{T} = \{ \{s_4\} \}$

$p = \{s_4\}$

$\alpha =$

```
P ← { F, S-F }
while ( P is still changing )
        T ← { }
        for each set p ∈ P
                T ← T ∪ Split (p)
        P ← T

Split (p)
        for each α ∈ ∑
        {
                if α splits p into p1 and p2
                then return {p1, p2}
        }
        return p;
```



Minimization of DFA

$P_0 = \{ \{s_4\}, \{s_0, s_1, s_2, s_3\} \}$

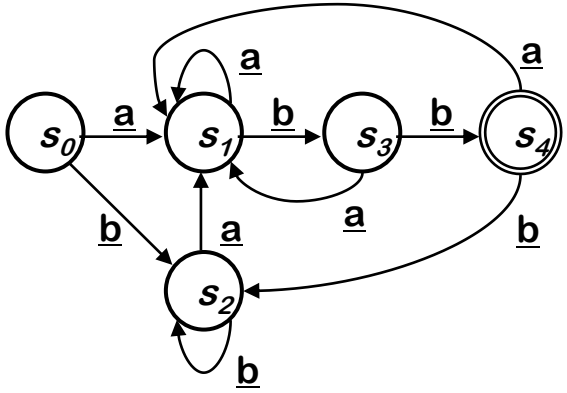
$\sum = \{a, b\}$

$T = \{ \{s_4\} \}$

$p = \{s_0, s_1, s_2, s_3\}$

$\alpha =$

```
P ← { F, S-F }
while ( P is still changing )
        T ← { }
        for each set p ∈ P
                T ← T ∪ Split (p)
        P ← T

Split (p)
        for each α ∈ ∑
        {
                if α splits p into p1 and p2
                then return {p1, p2}
        }
        return p;
```



Minimization of DFA

$\mathbf{P_0} = \{ \{s_4\}, \{s_0, s_1, s_2, s_3\} \}$

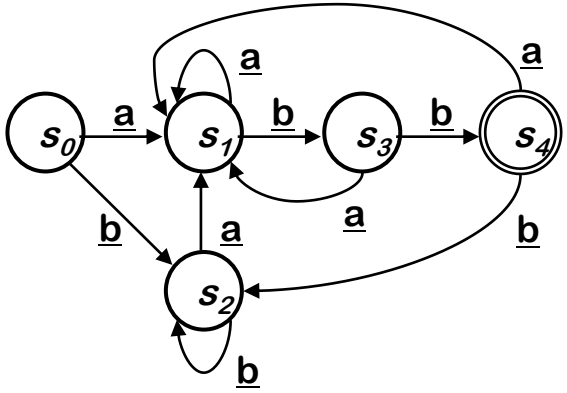
$\sum = \{a, b\}$

$\mathbf{T} = \{ \{s_4\} \}$

$p = \{s_0, s_1, s_2, s_3\}$

$\alpha =$

```
P ← { F, S-F }
while ( P is still changing )
        T ← { }
        for each set p ∈ P
                T ← T ∪ Split (p)
        P ← T

Split (p)
        for each α ∈ ∑
        {
                if α splits p into p1 and p2
                then return {p1, p2}
        }
        return p;
```



Minimization of DFA

$\mathbf{P_0}$ = { $\{s_4\}$, $\{s_0, s_1, s_2, s_3\}$ }

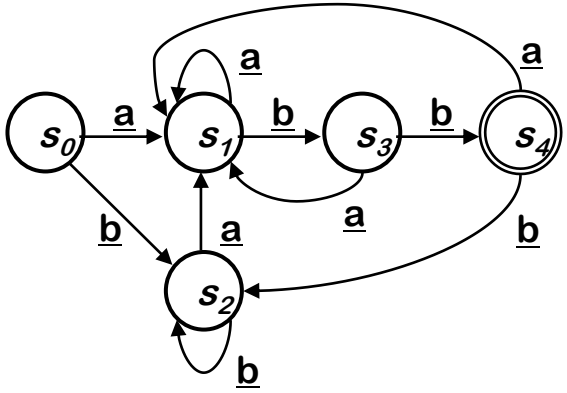
$\sum = \{\underline{a}, \underline{b}\}$

T = { $\{s_4\}$ }

p = $\{s_0, s_1, s_2, s_3\}$

α = **a**

```
P ← { F, S-F }
while ( P is still changing )
        T ← { }
        for each set p ∈ P
                T ← T ∪ Split (p)
        P ← T

Split (p)
        for each α ∈ ∑
        {
                if α splits p into p1 and p2
                then return {p1, p2}
        }
        return p;
```



Minimization of DFA

$P_0 = \{ \{s_4\}, \{s_0, s_1, s_2, s_3\} \}$

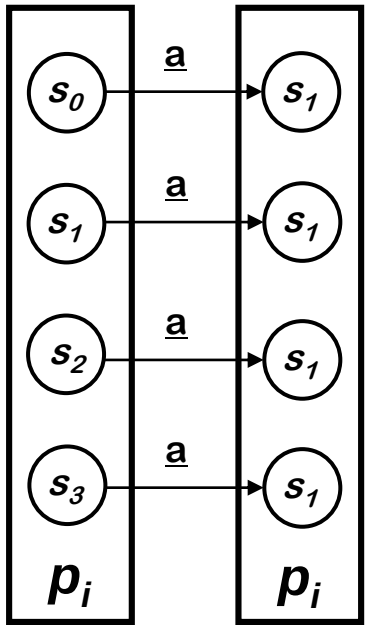
$\sum = \{a, b\}$

$T = \{ \{s_4\} \}$

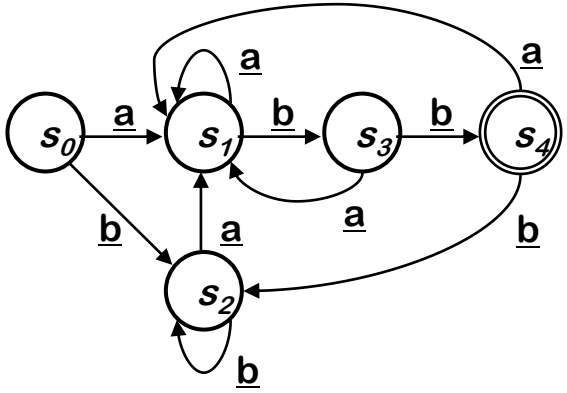
$p = \{s_0, s_1, s_2, s_3\}$

$\alpha = a$

```
P ← { F, S-F }
while ( P is still changing )
        T ← { }
        for each set p ∈ P
                T ← T ∪ Split (p)
        P ← T

Split (p)
        for each α ∈ ∑
        {
                if α splits p into p1 and p2
                then return {p1, p2}
        }
        return p;
```



- Split on **a**:
  - None…



Minimization of DFA

$\mathbf{P_0} = \{ \{s_4\}, \{s_0, s_1, s_2, s_3\} \}$

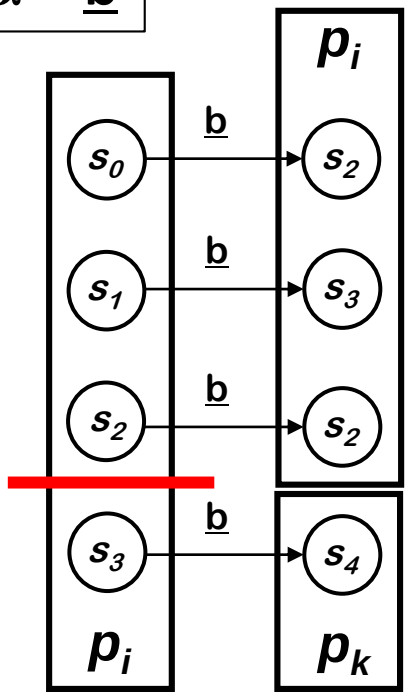
$\sum = \{a, b\}$

$\mathbf{T} = \{ \{s_4\} \}$

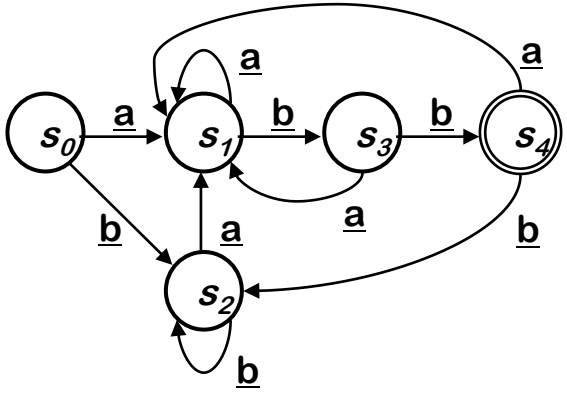
$p = \{s_0, s_1, s_2, s_3\}$

$\alpha = \underline{b}$

```
P ← { F, S-F }
while ( P is still changing )
        T ← { }
        for each set p ∈ P
                T ← T ∪ Split (p)
        P ← T

Split (p)
        for each α ∈ ∑
        {
                if α splits p into p1 and p2
                then return {p1, p2}
        }
        return p;
```



- Split on **b**:
  - α splits p_i into **$\{s_0, s_1, s_2\}$** and **$\{s_3\}$**



Minimization of DFA

$P_0 = \{ \{s_4\}, \{s_0, s_1, s_2, s_3\} \}$

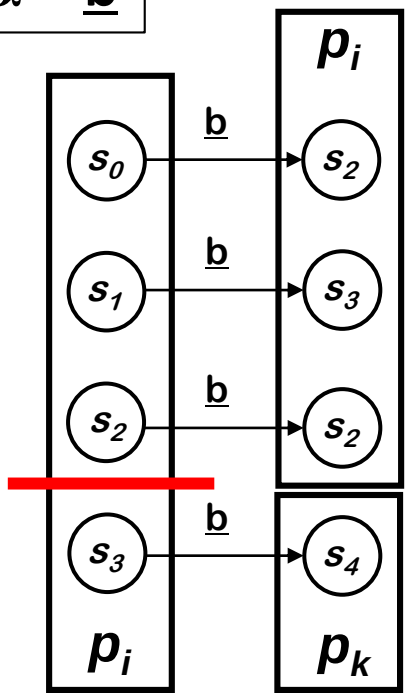
$\sum = \{a, b\}$

$T = \{ \{s_4\} \}$

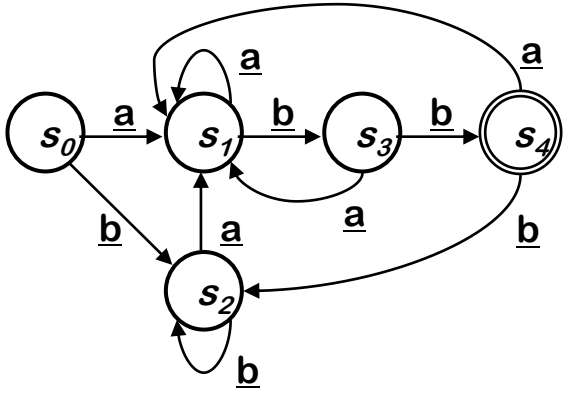
$p = \{s_0, s_1, s_2, s_3\}$

$\alpha = b$

```
P ← { F, S-F }
while ( P is still changing )
        T ← { }
        for each set p ∈ P
                T ← T ∪ Split (p)
        P ← T

Split (p)
        for each α ∈ ∑
        {
                if α splits p into p1 and p2
                then return {p1, p2}
        }
        return p;
```



- **Return** $\{\{s_0, s_1, s_2\}, \{s_3\}\}$



Minimization of DFA

$P_0 = \{ \{s_4\}, \{s_0, s_1, s_2, s_3\} \}$

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

$T = \{ \{s_4\} \}$

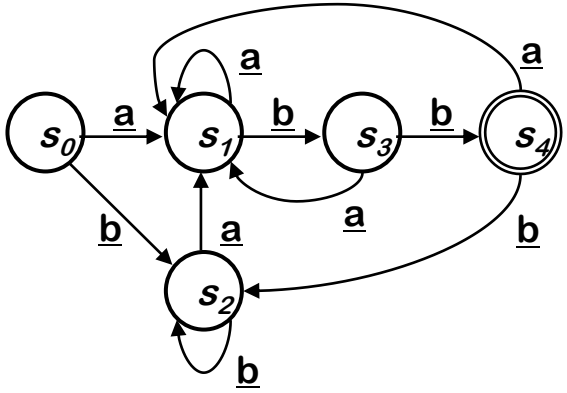
$p = \{s_0, s_1, s_2, s_3\}$

$\alpha =$

- **Split(p) =** $\{\{s_0,s_1,s_2\},\{s_3\}\}$

$T = \{ \{s_0,s_1,s_2\}, \{s_3\} \}$

```
P ← { F, S-F }
while ( P is still changing )
        T ← { }
        for each set p ∈ P
                T ← T ∪ Split (p)
        P ← T

Split (p)
        for each α ∈ ∑
        {
                if α splits p into p1 and p2
                then return {p1, p2}
        }
        return p;
```



Minimization of DFA

$P_0 = \{ \{s_4\}, \{s_0, s_1, s_2, s_3\} \}$

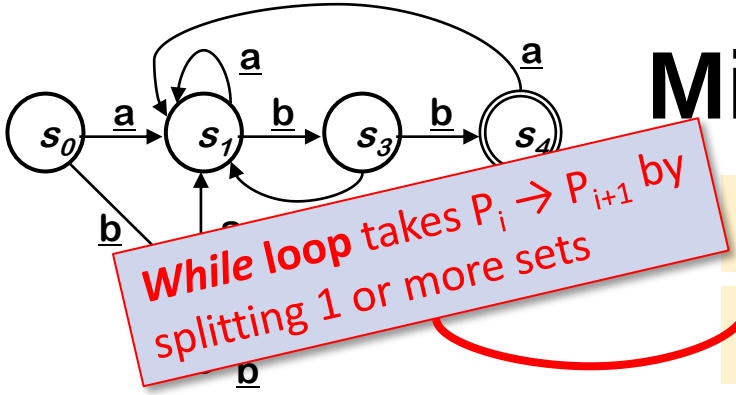
$\sum = \{a, b\}$

$T = \{ \{s_4\}, \{s_0, s_1, s_2\}, \{s_3\} \}$

$p = \{s_0, s_1, s_2, s_3\}$

$\alpha =$

```
P ← { F, S-F }
while ( P is still changing )
        T ← { }
        for each set p ∈ P
                T ← T ∪ Split (p)
        P ← T

Split (p)
        for each α ∈ ∑
        {
                if α splits p into p1 and p2
                then return {p1, p2}
        }
        return p;
```



Minimization of DFA

$P_1 = \{ \{s_4\}, \{s_0, s_1, s_2\}, \{s_3\} \}$

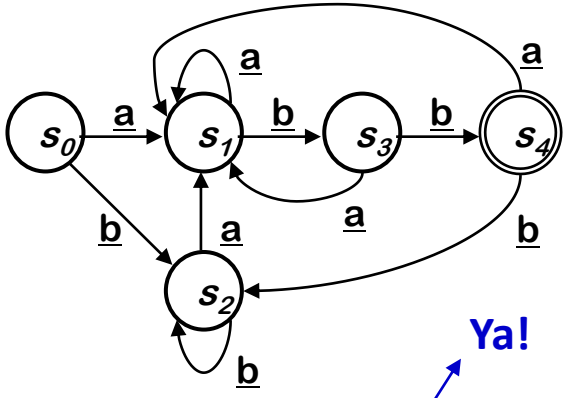
$T = \{ \{s_4\}, \{s_0, s_1, s_2\}, \{s_3\} \}$

$p = \{s_0, s_1, s_2, s_3\}$

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

$\alpha =$

```
P ← { F, S-F }
while ( P is still changing )
        T ← { }
        for each set p ∈ P
                T ← T ∪ Split (p)
        P ← T

Split (p)
        for each α ∈ ∑
        {
                if α splits p into p1 and p2
                then return {p1, p2}
        }
        return p;
```



Minimization of DFA

$P_1 = \{ \{s_4\}, \{s_0, s_1, s_2\}, \{s_3\} \}$

$T = \{ \{s_4\}, \{s_0, s_1, s_2\}, \{s_3\} \}$

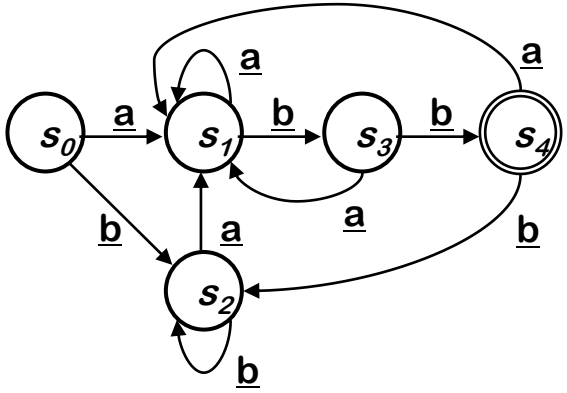
$p = \{ \}$

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

$\alpha =$

Ya!

```
P ← { F, S-F }
while ( P is still changing )
        T ← { }
        for each set p ∈ P
                T ← T ∪ Split (p)
        P ← T

Split (p)
        for each α ∈ ∑
        {
                if α splits p into p1 and p2
                then return {p1, p2}
        }
        return p;
```



Minimization of DFA

$P_1 = \{ \{s_4\}, \{s_0, s_1, s_2\}, \{s_3\} \}$

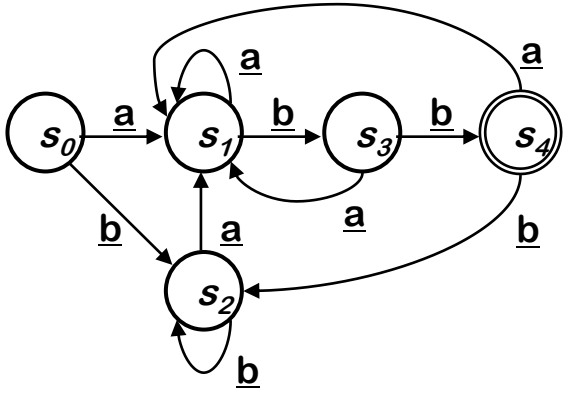
$T = \{ \}$

$p = \{ \}$

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

$\alpha =$

```
P ← { F, S-F }
while ( P is still changing )
        T ← { }
        for each set p ∈ P
                T ← T ∪ Split (p)
        P ← T

Split (p)
        for each α ∈ ∑
        {
                if α splits p into p1 and p2
                then return {p1, p2}
        }
        return p;
```



Minimization of DFA

$P_1 = \{ \{s_0, s_1, s_2\}, \{s_3\}, \{s_4\} \}$

Let's sort it by its subscript for readability!

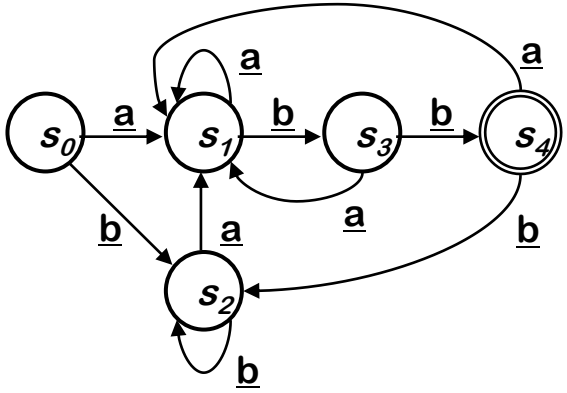
$T = \{ \}$

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

```
P ← { F, S-F }
while ( P is still changing )
        T ← { }
        for each set p ∈ P
                T ← T ∪ Split (p)
        P ← T

Split (p)
        for each α ∈ ∑
        {
                if α splits p into p1 and p2
                then return {p1, p2}
        }
        return p;
```

- **Split(p)**

	α = a	α = b
$p = \{s_0, s_1, s_2\}$	none	$\{s_0, s_2\}, \{s_1\}$
$p = \{s_3\}$	---	---
$p = \{s_4\}$	---	---



Minimization of DFA

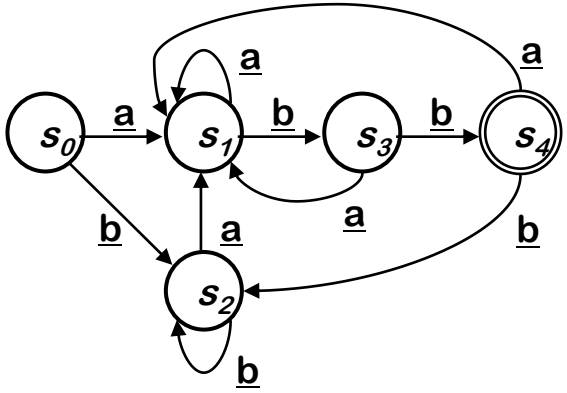
$P_1 = \{ \{s_0,s_1,s_2\}, \{s_3\}, \{s_4\} \}$

$T = \{ \{s_0,s_2\}, \{s_1\}, \{s_3\}, \{s_4\}\}$

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

$p = \{ \}$

```
P ← { F, S-F }
while ( P is still changing )
        T ← { }
        for each set p ∈ P
                T ← T ∪ Split (p)
        P ← T

Split (p)
        for each α ∈ ∑
        {
                if α splits p into p1 and p2
                then return {p1, p2}
        }
        return p;
```



Minimization of DFA

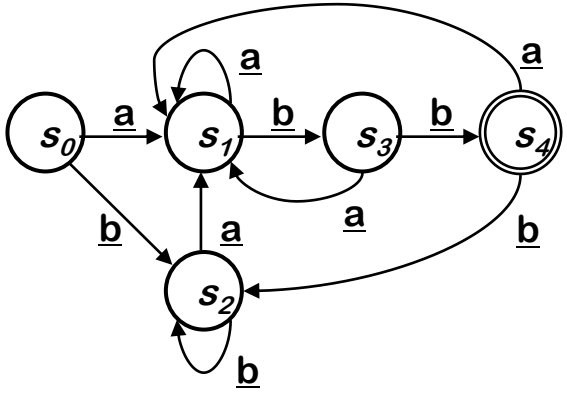
$P_2 = \{ \{s_0,s_2\}, \{s_1\}, \{s_3\}, \{s_4\} \}$

$T = \{ \{s_0,s_2\}, \{s_1\}, \{s_3\}, \{s_4\} \}$

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

$p = \{ \}$

```
P ← { F, S-F }
while ( P is still changing )
        T ← { }
        for each set p ∈ P
                T ← T ∪ Split (p)
        P ← T

Split (p)
        for each α ∈ ∑
        {
                if α splits p into p1 and p2
                then return {p1, p2}
        }
        return p;
```



Minimization of DFA

$P_2 = \{ \{s_0, s_2\}, \{s_1\}, \{s_3\}, \{s_4\} \}$

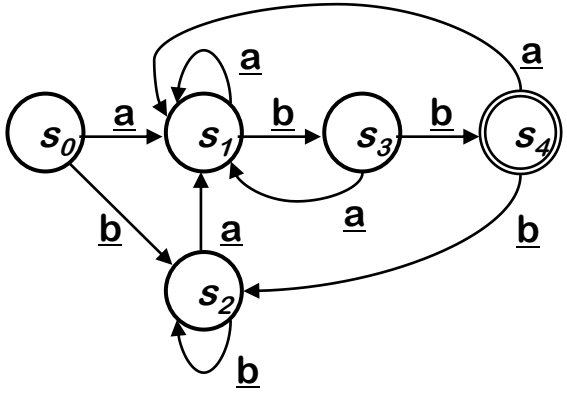
$T = \{ \}$

$p = \{s_0, s_2\}, \{s_1\}, \{s_3\}, \{s_4\}$

1st iter 2nd 3rd 4th

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

```
P ← { F, S-F }
while ( P is still changing )
        T ← { }
        for each set p ∈ P
                T ← T ∪ Split (p)
        P ← T

Split (p)
        for each α ∈ ∑
        {
                if α splits p into p1 and p2
                then return {p1, p2}
        }
        return p;
```



Minimization of DFA

$P_2 = \{ \{s_0, s_2\}, \{s_1\}, \{s_3\}, \{s_4\} \}$

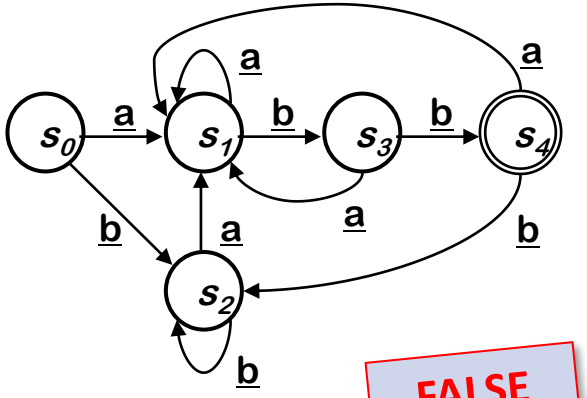
$T = \{ \{s_0, s_2\}, \{s_1\}, \{s_3\}, \{s_4\} \}$

$p = \{s_0, s_2\}, \{s_1\}, \{s_3\}, \{s_4\}$

1st iter 2nd 3rd 4th

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

```
P ← { F, S-F }
while ( P is still changing )
        T ← { }
        for each set p ∈ P
                T ← T ∪ Split (p)
        P ← T

Split (p)
        for each α ∈ ∑
        {
                if α splits p into p1 and p2
                then return {p1, p2}
        }
        return p;
```



Minimization of DFA

$P_2 = \{ \{s_0, s_2\}, \{s_1\}, \{s_3\}, \{s_4\} \}$

$T = \{ \{s_0, s_2\}, \{s_1\}, \{s_3\}, \{s_4\} \}$

$p = \{s_0, s_2\}, \{s_1\}, \{s_3\}, \{s_4\}$

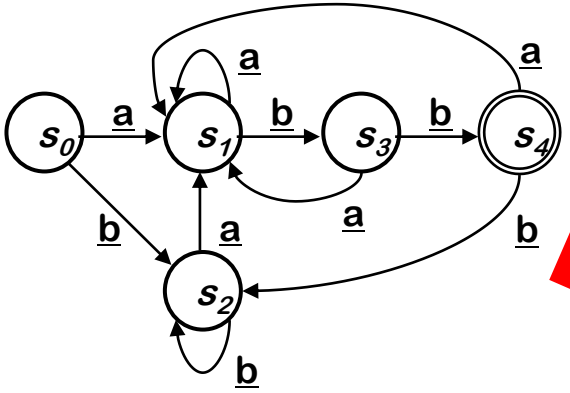
1st iter 2nd 3rd 4th

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

FALSE (P_2 == T)

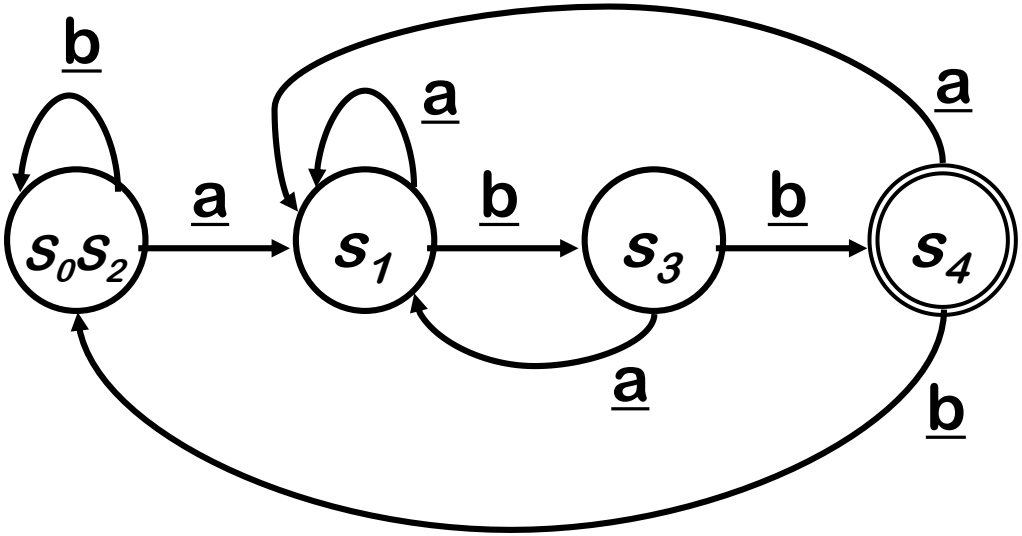
```
P ← { F, S-F }
while ( P is still changing )
        T ← { }
        for each set p ∈ P
                T ← T ∪ Split (p)
        P ← T

Split (p)
        for each α ∈ ∑
        {
                if α splits p into p1 and p2
                then return {p1, p2}
        }
        return p;
```

- While loop comes to a halt…
  - P does not be changed by previous step



Minimization of DFA

$$P = \{ \{s_0, s_2\}, \{s_1\}, \{s_3\}, \{s_4\} \}$$



DFA Minimization

The algorithm:

```
P ← { F, S-F }
while ( P is still changing )
    T ← { }
    for each set p ∈ P
        T ← T ∪ Split (p)
    P ← T

Split (p)
    for each α ∈ ∑
    {
        if α splits p into p1 and p2
        then return {p1, p2}
    }
    return p;
```

This is a fixed-point algorithm!

Why does this **terminate**?

- $p \in 2^S$ (powerset)
- Start off with 2 subsets of S: *F and S-F*
- ***While* loop** takes $P_i \rightarrow P_{i+1}$ by splitting 1 or more sets
- P_{i+1} is at least one step closer to the partition with |S| sets
- Maximum of |S| splits

Note that

- Partitions are never combined, only split.

→ algorithm eventually terminates

DFA Minimization

The algorithm:

```
P ← { F, S-F }
while ( P is still changing )
    T ← { }
    for each set p ∈ P
        T ← T ∪ Split (p)
    P ← T

Split (p)
    for each α ∈ ∑
    {
        if α splits p into p1 and p2
        then return {p1, p2}
    }
    return p;
```

This is a fixed-point algorithm!

Why does this **work**?

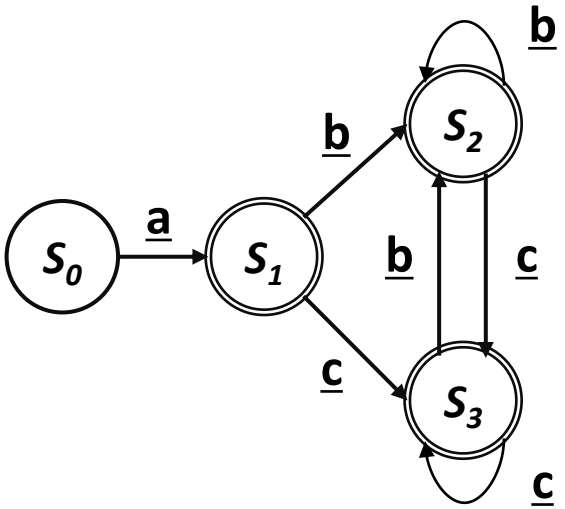
- *The algorithm maintains 2 invariants:*
  - 1) States remaining in the same set have not been distinguished yet by any string.
  - 2) States winding up in different sets are distinguishable by some string.
- The sets in the final partition contains the set of distinguishable states of the DFA.
- Proof sketch: Dragon book, 2nd ed., Section 3.9.6.

DFA Minimization Example

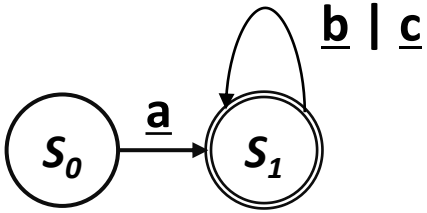
- Applying the minimization algorithm…

	Current Partition	Split on		
		a	b	c
P_0	$\{S_1, S_2, S_3\}\{S_0\}$	*none*	*none*	*none*

accepting states



… to produce the minimal DFA

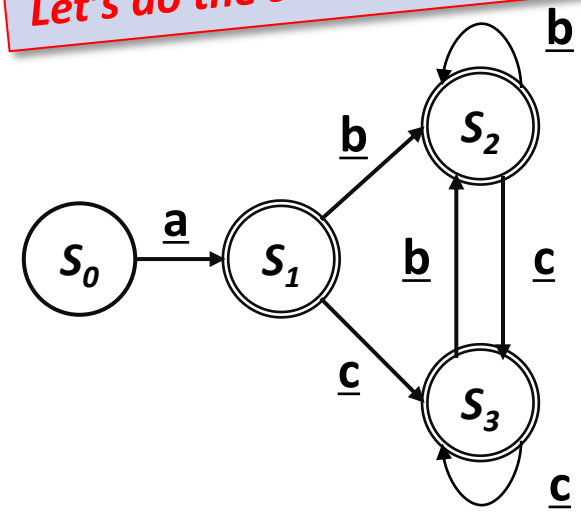


In lecture 5, we observed that a human would design a simpler automaton than Thompson's construction & the subset construction did.

Minimizing that DFA produces the one that a human would design.

Theoretical result: every RE language can be recognized by a minimal-state DFA that is unique up to state names

Let's do the same things to this DFA!



Minimization of DFA

P = { }

T = { }

p =

$\sum$ = {a, b, c}

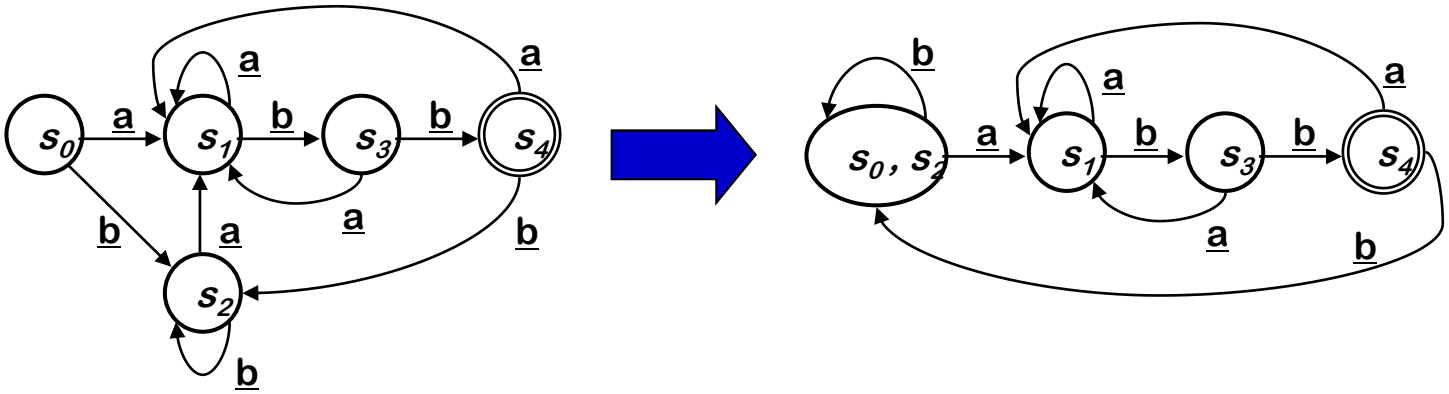
P ← { *F, S-F* }
while (*P* is still changing)
    T ← { }
    for each set *p* ∈ *P*
        T ← *T* ∪ Split (*p*)
    P ← *T*

Split (*p*)
    for each *α* ∈ ∑
    {
        if *α* splits *p* into *p1* and *p2*
        then return *{p1, p2}*
    }
    return *p*;

A Detailed Example

- Applying the minimization algorithm to the DFA for **(a|b)*abb**

Partition	Set	Split on a	Split on b
P_0	$\{s_0,s_1,s_2,s_3\}$	none	$\{s_0,s_1,s_2\},\{s_3\}$
	$\{s_4\}$	---	---
P_1	$\{s_0,s_1,s_2\}$	none	$\{s_0,s_2\}, \{s_1\}$
	$\{s_3\}$	---	---
	$\{s_4\}$	---	---
P_2	$\{s_0,s_2\},$	none	none
	$\{s_1\}$	---	---
	$\{s_3\}$	---	---
	$\{s_4\}$	---	---



Outline

- The role of a scanner ✓
- Scanner concepts ✓
  - Tokens, Lexemes, Patterns
- Regular Expression & Automata (to be continued…)
  - Definitions of REs, DFAs and NFAs ✓
  - REs→NFA ✓
    (Thompson's construction, Algorithm 3.3, Red Dragon book, Algorithm 3.23, Purple Dragon book)
  - NFA→DFA ✓
    (subset construction, Algorithm 3.2, Red Dragon book, Algorithm 3.20, Purple Dragon)
  - DFA→minimal-state DFA ✓
    (state minimization, Algorithm 3.6, Red Dragon book, Algorithm 3.39, Purple Dragon book)
- Scanner generators

Scanner Generators

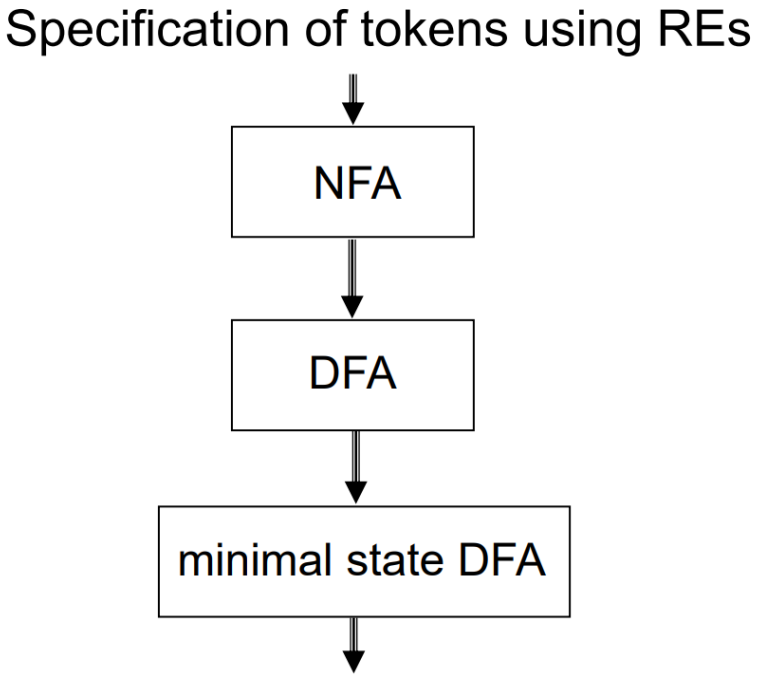
- Scanners generated in C:
  - lex (UNIX)
  - flex (GNU's fast lex, UNIX)
  - Mks lex (MS-DOS, Windows, OS/2)

- Scanners generated in Java:
  - Jlex (Princeton University)
  - JFlex
  - JavaCC (SUN Microsystems → now, Oracle)

The Scanner Spec in JLex

```
user code // copied verbatim to the scanner file
%%
JLEX directives (declarations)
%%
regular expression rules
```

How a Scanner Generator Works



- Table-driven code (JLex), simulates a DFA on an input
- Hard-wired code
  (like our hand-crafted scanner for Assignment #1)
  - Assignment #1 = Scanner + Parser

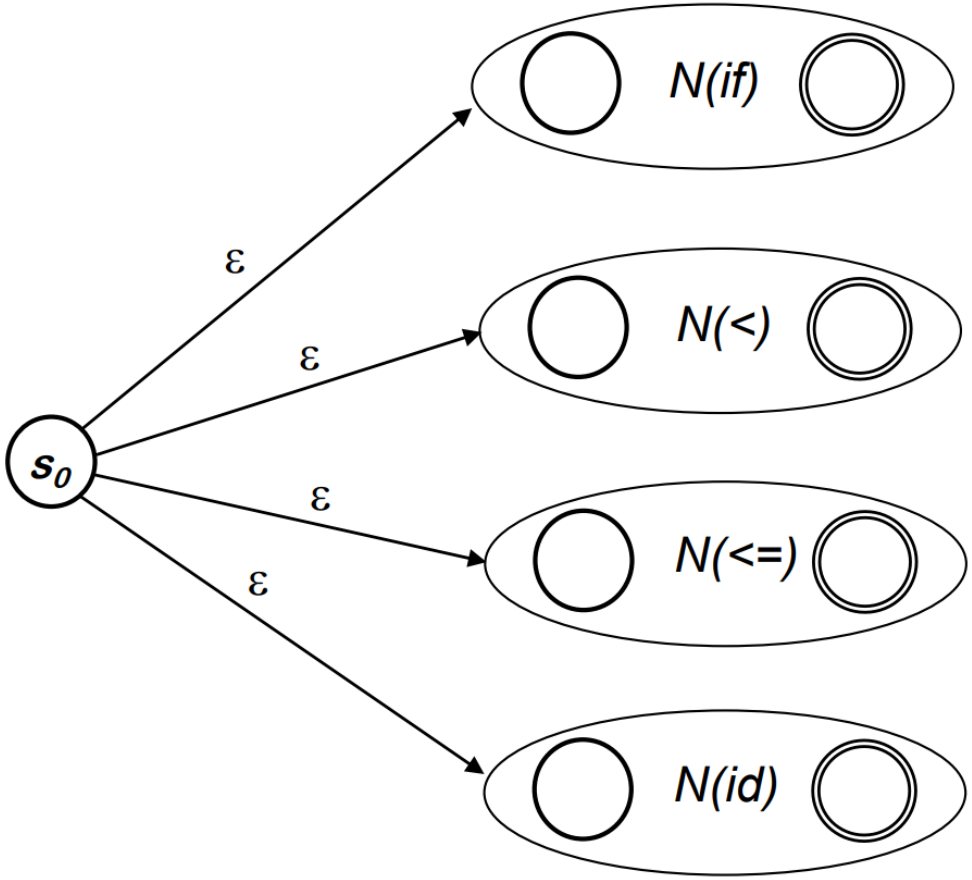
JLex Example Spec

```
%%
LETTER=[a-zA-Z_]
DIGIT=[0-9]
%%
"if" { return new Token(Token.IF, "if", src_pos); }
"<"  { return new Token(Token.LESS, "<", src_pos); }
"<="
  { return new Token(Token.LESS_EQ, "<=", src_pos); }
{LETTER}({LETTER}|{DIGIT})*
  { return new Token.ID, "spelling", src_pos); }
```

Two rules:

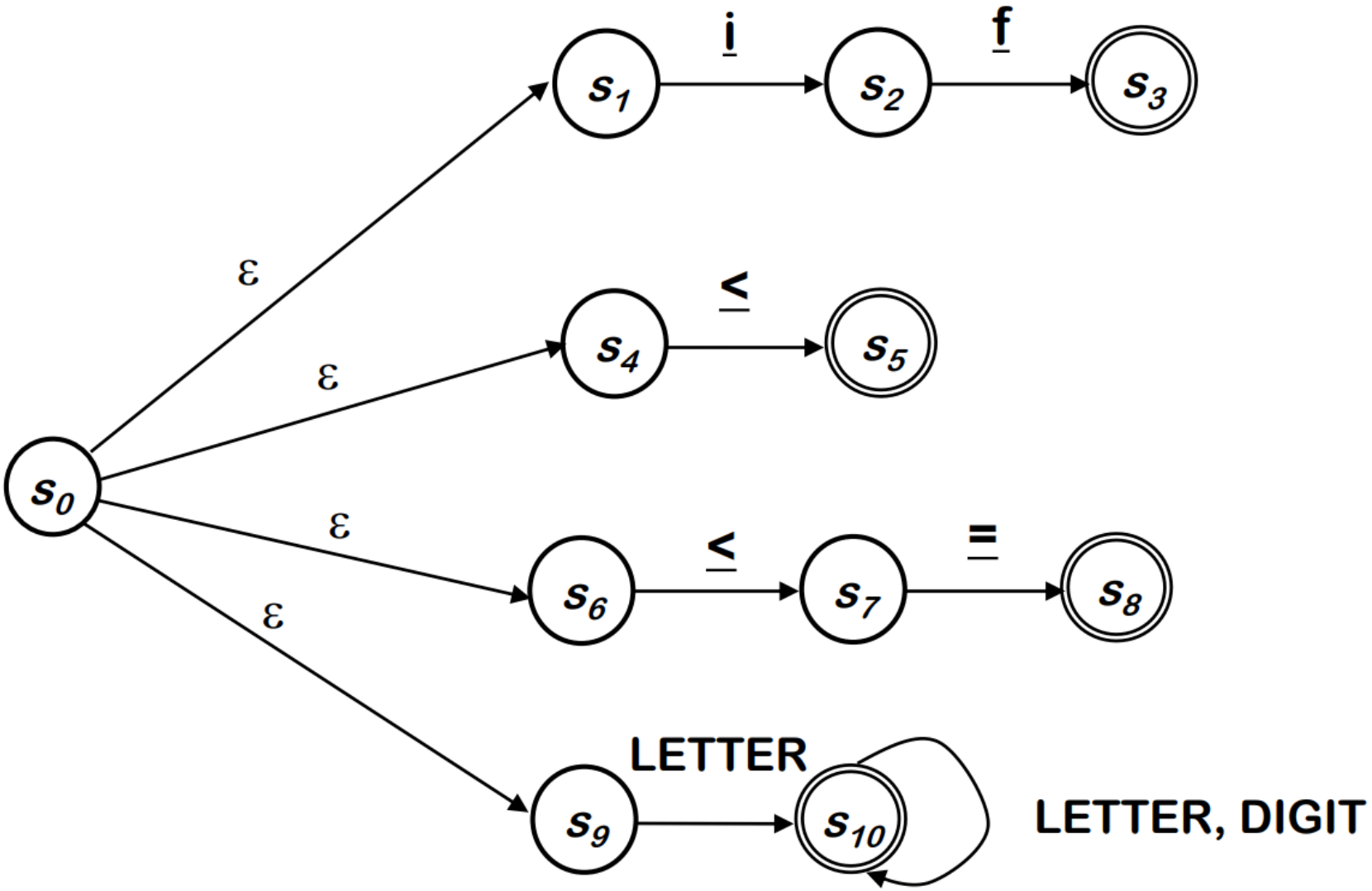
- The first pattern is used if more than one pattern matches.
  - Example: **if** as a keyword (IF) and not as an identifier (ID).
- The longest possible match is taken.
  - Example: “<=“ as one token.

NFA for JLex Example Spec



- The NFAs for the different REs are combined as above.
- Instead of an NFA, a DFA can also be used for each pattern.

NFA for JLex Example Spec



Running JLex on a Sample Scanner Spec

Scanner.l: the spec passed to the scanner generator JLex

```
java JLex.Main Scanner.l
```

```
Processing first section -- user code.
Processing second section -- JLex declarations.
Processing third section -- lexical rules.
Creating NFA machine representation.
NFA comprised of 315 states.
Working on character classes.::.::::.:.::.::.:.:.:...
NFA has 46 distinct character classes.
Creating DFA transition table.
Working on DFA states.................................
Minimizing DFA transition table.
109 states after removal of redundant states.
Outputting lexical analyzer code.
```

Scanner.l.java: the generated scanner

```
javac Scanner.l.java
```

Limitations of Regular Languages

Advantages of Regular Expressions

- Simple & powerful notation for specifying patterns
- Automatic construction of fast recognizers (scanners)
- Many patterns can be specified with Res

Example — an expression grammar

Term: [a-zA-Z] ([a-zA-z] | [0-9])*

Op: + | - | * | /

Expr : (Term Op)* Term

Of course, this would generate a DFA …

If REs are so useful …

Why not use them for everything?

Limitations of Regular Languages (cont.)

- Regular expressions are limited in what can be expressed.
  - Example: it is not possible to specify a regular expression for $\underline{0}^n\underline{1}^n$ (the set of strings of N zeroes followed by N ones).

    For the same reason, we cannot specify the set of arithmetic expressions for which left and right parentheses match.
    e.g., (a), ((a)), (((a))), ((((a))))

- Regular expressions only usable to specify keywords, identifiers, literals, operators and punctuation characters of a programming language. For everything else (expressions, statements, nested statements, …) we need a more powerful mechanism

Regular Expressions and Context-Free Grammars

- The “languages” that can be defined by REs and Context-free grammars (CFGs) have been extensively studied by theoretical computer scientists. These are some important conclusions / terminologies:
- RE are a “weaker” formalism than CFGs: Any language expressible by a RE can be expressed by a CFG **but not the other way around**!
- The languages expressible as RE are called regular languages
- Generally: a language that exhibits “self embedding” cannot be expressed by a RE.
- Programming languages exhibit self embedding.
  Example: an expression can contain another expression

```
expr ::=  id | integer | - expr | ( expr ) | expr op expr
op   ::=  +  |  -  |  *  |  /
```

How many expressions can be derived? Infinitely many.

Table-driven vs. Direct-coded Scanners

Jeonggeun Kim
Kyungpook National University

This lecture's materials are based on Prof. Bernd Burgstaller's slides (CSI4104 Compiler Design, Yonsei University)

Table-driven vs. Directed-coded Scanners

δ	r	0,1,2,3 ,4,5,6, 7,8,9	*All others*
s_0	s_1	s_e	s_e
s_1	s_e	s_2	s_e
s_2	s_e	s_2	s_e
s_e	s_e	s_e	s_e

Table encoding RE

```
Char ← next character
State ← s0

while (Char ≠ EOF)
    State ← δ(State,Char)
     Char ← next character

if (State is a final state )
    then report success
    else  report failure
```

Skeleton recognizer

Table-driven

```
switch (currentChar) {
   case '+': return token representation for '+'
   case '<':
                          takeIt();   // adds '<' to the current lexeme
                          if (currentChar == '=')
                                        return token representation for '<='
                          else
                                        return token representation for '<'
```

Direct-coded

Ease of Implementation & Performance

Table-driven scanner encodes the automaton as a big table

- Costly lookup (computation of table cell based on state and input character); not cache-friendly if tablesize exceeds size of cache.
- Easy, automated way to implement. Hard to implement manually.

Direct-coded scanner encodes the automaton in the **program counter:**

```
"if '<' then read next char and check for "<="
else ..."
```

- Automated and manual implementations feasible
- No table and thus no table lookup needed
- More complex control flow
  - Ok for architectures with good branch prediction (x86, ...).
- Tends to be faster than table-driven scanner on general-purpose CPU

Training for Lexical Analysis

Jeonggeun Kim
Kyungpook National University

This lecture's materials are based on Prof. Bernd Burgstaller's slides (CSI4104 Compiler Design, Yonsei University)

Writing Regular Expressions

Write a regular expression for each of the following sets of tokens:

1) Ruby binary literals consisting of "0b" followed by the binary number. Examples: 0b001011, 0b01.
2) Ruby binary literals, with an optional underscore ("_") between a pair of binary digits.
   Examples: 0b0_101 , 0b11_01 , but not 0b_1 and not 0b1_ .
3) Ada identifiers: a letter followed by any number of letters, digits, and underlines. An identifier must not end in an underline or have two underlines in a row.
4) A floating-point number: one or more digits (whole-number part) followed by a decimal point (".") and one or more digits (fractional part).
   Examples: 2.24, 0.1234.
5) Floating-point number in scientific notation: same as above, but optionally followed by "e" or "E", and a signed integer exponent.
   Examples: 1.2e-2, 2.3E+34, 2.3E34.

1) Ruby binary literals consisting of “0b” followed by the binary number.
   Examples: 0b001011, 0b01.

2) Ruby binary literals, with an optional underscore (“_”) between a pair of binary digits.
   Examples: `0b0_101` , `0b11_01` , but not `0b_1` and not `0b1_` .

3) Ada identifiers: a letter followed by any number of letters, digits, and underlines. An identifier must not end in an underline or have two underlines in a row.

4) A floating-point number: one or more digits (whole-number part) followed by a decimal point (“.”) and one or more digits (fractional part).
   Examples: 2.24, 0.1234.

5) Floating-point number in scientific notation: same as above, but optionally followed by “e” or “E”, and a signed integer exponent.
   Examples: 1.2e-2, 2.3E+34, 2.3E34.

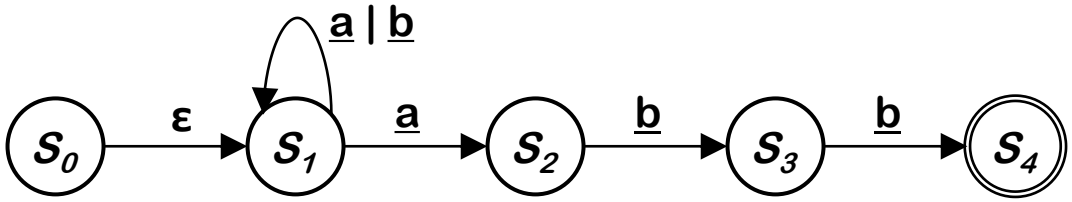
Subset Construction Algorithm (Animation)

Jeonggeun Kim
Kyungpook National University

This lecture's materials are based on Prof. Bernd Burgstaller's slides (CSI4104 Compiler Design, Yonsei University)

Example: Simulating an NFA ``on the fly''

Input = ...



- **On-the-fly simulation** of an NFA is useful in scenarios where a regular expression is used only once (example: in a text editor or IDE).
- In a compiler, the regular expressions for the programming language's tokens are used over and over again → need a more efficient approach → next slide.

Algorithm: NFA →DFA with Subset Construction

Subset construction works on sets of NFA states.

Each set of NFA states becomes a DFA state.

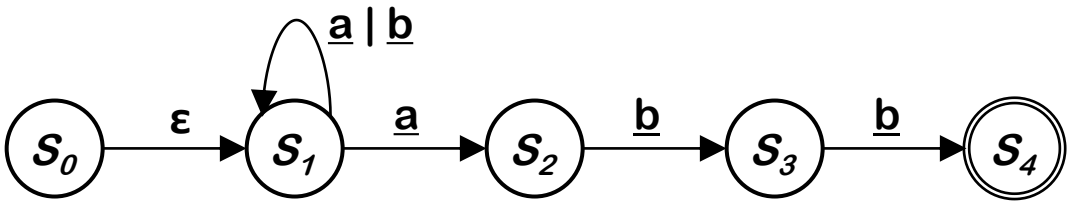
Two key functions

- *Move* (*S* , *a*) : set of states reachable from set *S* by a
- *ε-closure* (*S*) : set of states reachable from set *S* by ε

The algorithm:

- Start state derived from s_0 of the NFA:
  - Take its ε-closure S_0 = ε-closure($\{s_0\}$)
- Compute Move(S_0, α) for each $\alpha \in \Sigma$, and take its ε-closure
- Iterate until no more states are added

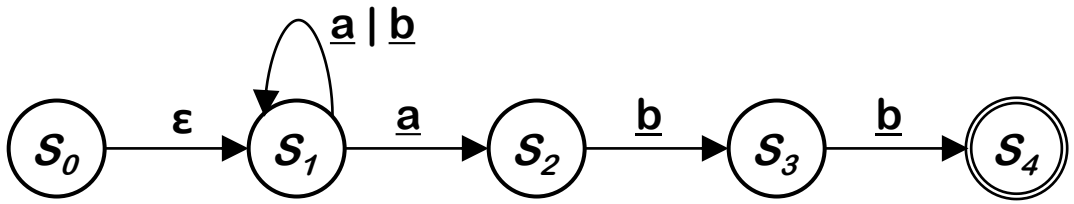
Sounds more complex than it is...




```
Dstates ← {};
add ε-closure(s0) as an
      unmarked state to Dstates;
while ( there is an unmarked
     state T in Dstates ) {
  mark T;
  for each α ∈ ∑ {
       U ← ε-closure(Move(T, α))
       if ( U ∉ Dstates ) then
          add U as an unmarked
                      state to Dstates;
       δ [T, α] ← U
  }
}
```

Note:

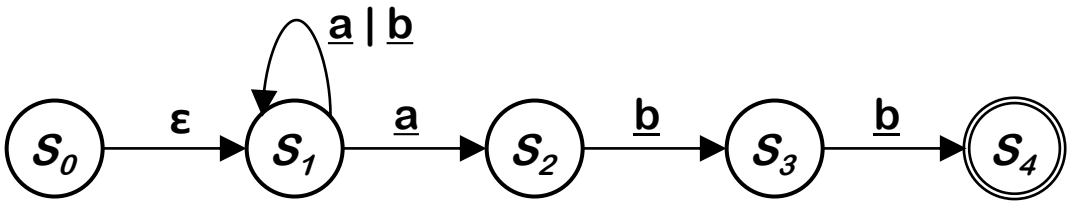
- here δ refers to the transition function of the DFA.
- δ takes a set of NFA-states (= a DFA state) and a symbol α as input, and returns a set of NFA-states (= a DFA state).



Dstates = { }

Dstates ← {};
add *ε-closure(s_0) as an unmarked state to Dstates;*
while *(there is an unmarked state T in Dstates) {*
  mark *T;*
  for each *α ∈ ∑ {*
    U ← ε-closure(Move(T, α))
    if *(U ∉ Dstates)* ***then***
      add *U as an unmarked state to Dstates;*
    δ [T, α] ← U
  }
}

ε-closure(s_0) = { s_0, s_1 }



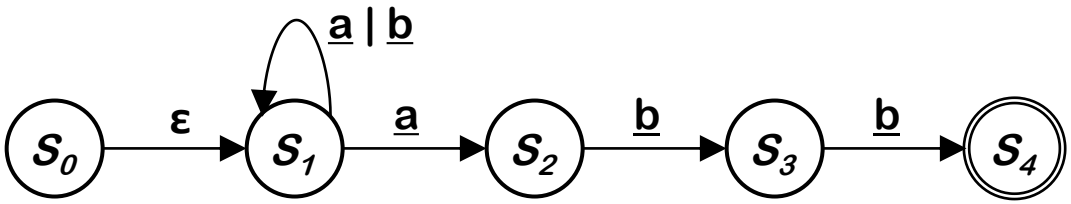
***Dstates* = { { s_0, s_1 } }**

Dstates ← {};
add *ε-closure(s_0) as an unmarked state to Dstates;*
while *(there is an unmarked state T in Dstates) {*
  mark *T;*
  for each *α ∈ ∑ {*
    U ← ε-closure(Move(T, α))
    if *(U ∉ Dstates)* ***then***
      add *U as an unmarked state to Dstates;*
    δ [T, α] ← U
  }
}

Dstates contains now one element, i.e., the state { s_0, s_1 } .

Note1: unmarked states are printed in red.

Note2: Dstates means ``deterministic states, the states that will make up the DFA''.

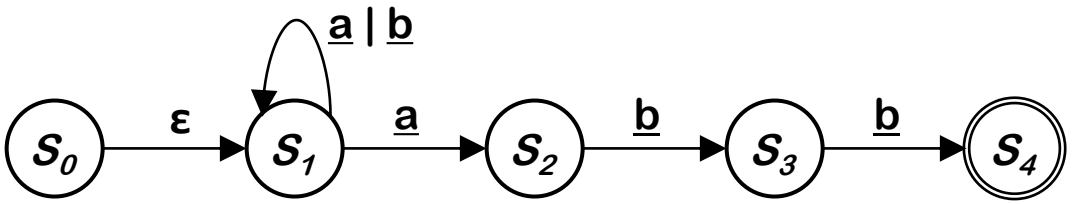


***Dstates* = { { s_0, s_1 } }**

```
Dstates ← {};
add ε-closure(s0) as an
      unmarked state to Dstates;
while ( there is an unmarked
     state T in Dstates ) {
  mark T;
  for each α ∈ ∑ {
       U ← ε-closure(Move(T, α))
       if ( U ∉ Dstates ) then
          add U as an unmarked
                       state to Dstates;
       δ [T, α] ← U
   }
}
```

Now we pick state { s_0, s_1 } from Dstates and mark it.

Note: marked states are printed in green.

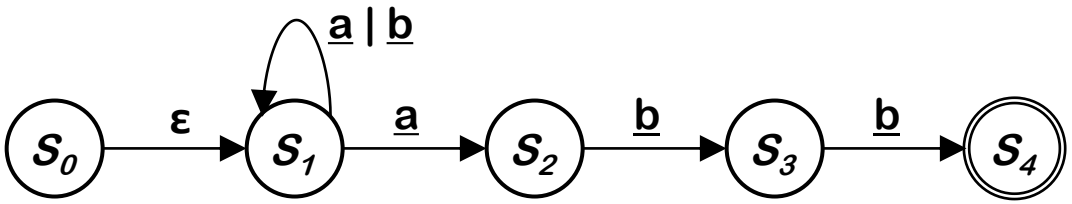


Dstates = { { s_0, s_1 } }

T = { s_0, s_1 } ∑ = { a, b }

```
Dstates ← {};
add ε-closure(s0) as an
      unmarked state to Dstates;
while ( there is an unmarked
    state T in Dstates ) {
  mark T;
  for each α ∈ ∑ {
       U ← ε-closure(Move(T, α))
       if ( U ∉ Dstates ) then
          add U as an unmarked
                       state to Dstates;
       δ [T, α] ← U
   }
}
```

The **for** loop is going to iterate over all symbols in ∑.
First iteration: a
Second iteration: b.




```
Dstates ← {};
add ε-closure(s0) as an
      unmarked state to Dstates;
while ( there is an unmarked
     state T in Dstates ) {
  mark T;
  for each α ∈ ∑ {
       U ← ε-closure(Move(T, α))
       if ( U ∉ Dstates ) then
          add U as an unmarked
                       state to Dstates;
       δ [T, α] ← U
   }
}
```

***Dstates* = { { s_0, s_1 } }**

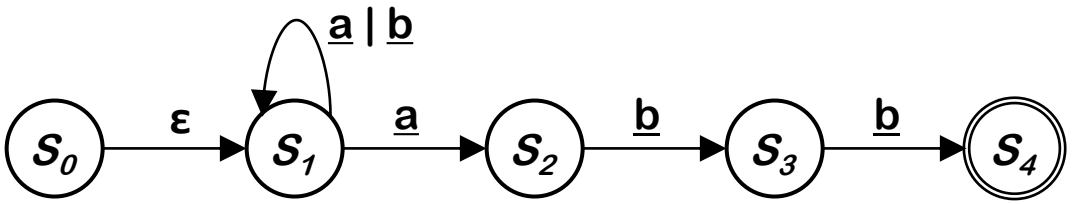
***T* = { s_0, s_1 }** ∑ = { a, b }

The **for** loop is going to iterate over all symbols in ∑.
First iteration: α = a
Second iteration: α = b.

Move (T, a) = { s_1, s_2 }

ε-closure({ s1, s2 }) = { s_1, s_2 }

U= { s_1, s_2 }




```
Dstates ← {};
add ε-closure(s0) as an
      unmarked state to Dstates;
while ( there is an unmarked
     state T in Dstates ) {
  mark T;
  for each α ∈ ∑ {
       U ← ε-closure(Move(T, α))
       if ( U ∉ Dstates ) then
          add U as an unmarked
                        state to Dstates;
       δ [T, α] ← U
   }
}
```

Dstates = { { s_0, s_1 } }

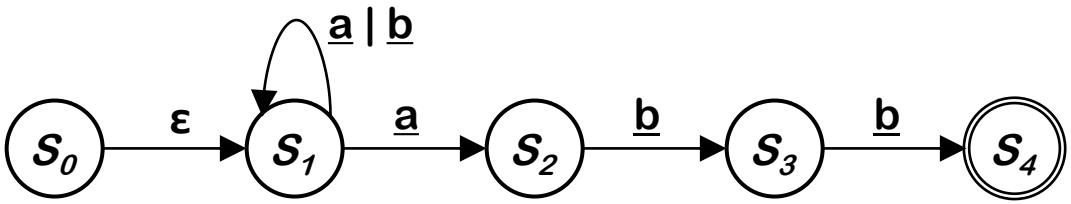
T = { s_0, s_1 }

∑ = { a, b }

The **for** loop is going to iterate over all symbols in ∑.
First iteration: α =a
Second iteration: α =b.

U= { s_1, s_2 }

State U is not contained in Dstates yet, so we add it as an unmarked state.



Dstates = { { s_0, s_1 } }

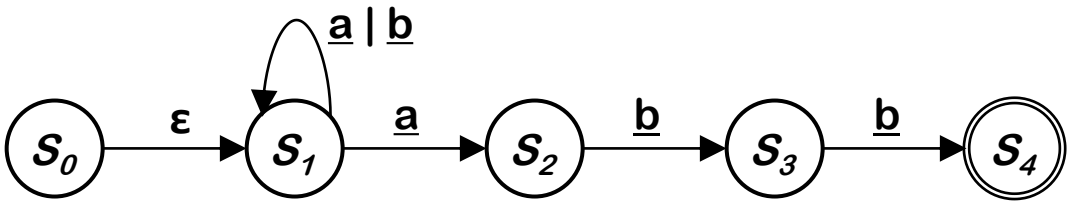
T = { s_0, s_1 } ∑ = { a, b } *U* = { s_1, s_2 }

```
Dstates ← {};
add ε-closure(s0) as an
      unmarked state to Dstates;
while ( there is an unmarked
     state T in Dstates ) {
  mark T;
  for each α ∈ ∑ {
       U ← ε-closure(Move(T, α))
       if ( U ∉ Dstates ) then
          add U as an unmarked
                        state to Dstates;
       δ [T, α] ← U
   }
}
```

Here we start assembling the transition function δ for our DFA.

If we read an a in state T, then the DFA will move to DFA state {s_1, s_2} :

δ	a	b
{s_0 ,s_1}	{s_1, s_2}	



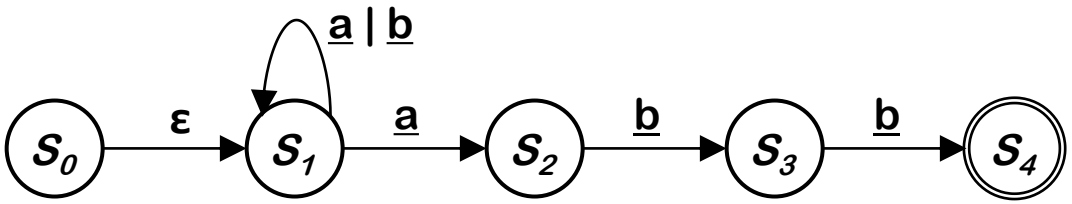
Dstates ← {};
add ε-closure(s_0) as an unmarked state to Dstates;
while (there is an unmarked state T in Dstates) {
  mark T;
  for each α ∈ ∑ {
    U ← ε-closure(Move(T, α))
    if (U ∉ Dstates) **then**
      add U as an unmarked state to Dstates;
    δ [T, α] ← U
  }
}

Dstates = { { s_0, s_1 }, { s_1, s_2 } }

T = { s_0, s_1 } ∑ = { a, b } U = { s_1, s_2 }

Second iteration of the **for** loop: α = b.

δ	a	b
{s_0 ,s_1}	{s_1, s_2}	




```
Dstates ← {};
add ε-closure(s0) as an
      unmarked state to Dstates;
while ( there is an unmarked
     state T in Dstates ) {
  mark T;
  for each α ∈ ∑ {
       U ← ε-closure(Move(T, α))
       if ( U ∉ Dstates ) then
          add U as an unmarked
                        state to Dstates;
       δ [T, α] ← U
   }
}
```

Dstates = { { s_0, s_1 }, { s_1, s_2 } }

T = { s_0, s_1 }

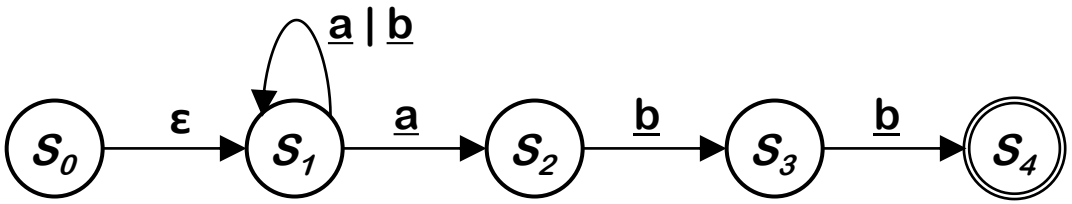
∑ = { a, b }

Second iteration of the **for** loop: α = b.

Move (T, b) = { s_1 }

ε-closure({ s1 }) = { s_1 }

U= { s_1 }




```
Dstates ← {};
add ε-closure(s0) as an
      unmarked state to Dstates;
while ( there is an unmarked
     state T in Dstates ) {
  mark T;
  for each α ∈ ∑ {
       U ← ε-closure(Move(T, α))
       if ( U ∉ Dstates ) then
          add U as an unmarked
                       state to Dstates;
       δ [T, α] ← U
   }
}
```

Dstates = { { s_0, s_1 }, { s_1, s_2 } }

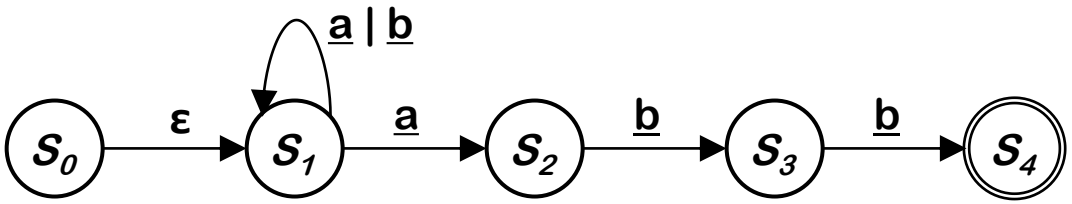
T = { s_0, s_1 }

∑ = { a, b }

Second iteration of the **for** loop: α = b.

U = { s_1 }

State U is not contained in Dstates yet, so we add it as an unmarked state.



Dstates ← {};
add ε-closure(s_0) as an unmarked state to Dstates;
while (there is an unmarked state T in Dstates) {
  mark T;
  for each α ∈ ∑ {
    U ← ε-closure(Move(T, α))
    if (U ∉ Dstates) **then**
      add U as an unmarked state to Dstates;
    δ [T, α] ← U
  }
}

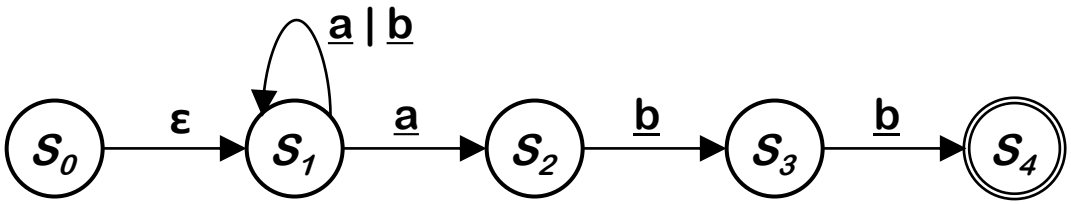
Dstates = { { s_0, s_1 }, { s_1, s_2 } }

T = { s_0, s_1 } ∑ = { a, b } U= { s_1 }

Second iteration of the **for** loop: α =b.

If we read a b in state T, then the DFA will move to state {s_1} :

δ	a	b
{s_0 ,s_1}	{s_1, s_2}	{ s_1 }



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

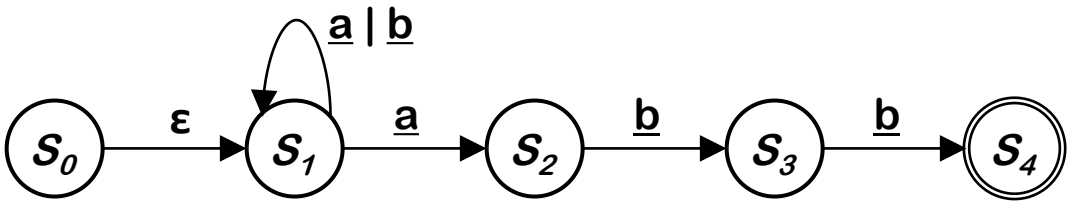
Dstates = **{** { s_0, s_1 }, { s_1, s_2 }, { s_1 } **}**

Dstates ← {};
add *ε-closure(s_0) as an unmarked state to Dstates;*
while *(there is an unmarked state T in Dstates) {*
  mark *T;*
  for each *α ∈ ∑ {*
    U ← ε-closure(Move(T, α))
    if *(U ∉ Dstates)* ***then***
      add *U as an unmarked state to Dstates;*
    δ [T, α] ← U
  }
}

Dstates contains now two unmarked states, { *s1, s2* } and { *s1* }.

We pick state { *s1, s2* } next.

δ	a	b
{s_0 ,s_1}	{s_1, s_2}	{ s_1 }



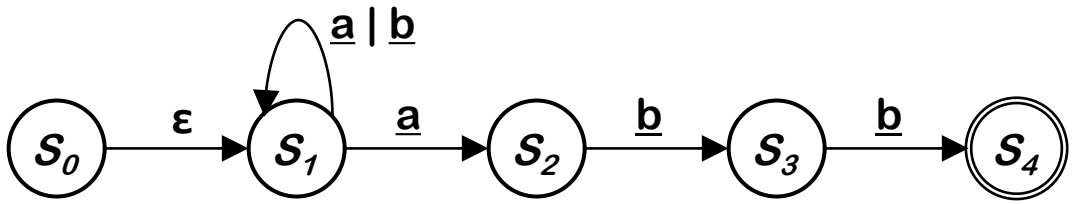
$\sum$ = { a, b }

Dstates = { { s_0, s_1 }, { s_1, s_2 }, { s_1 } }

T = { s_1, s_2 }

```
Dstates ← {};
add ε-closure(s0) as an
      unmarked state to Dstates;
while ( there is an unmarked
     state T in Dstates ) {
  mark T;
  for each α ∈ ∑ {
       U ← ε-closure(Move(T, α))
       if ( U ∉ Dstates ) then
          add U as an unmarked
                       state to Dstates;
       δ [T, α] ← U
   }
}
```

δ	a	b
{s_0 ,s_1}	{s_1, s_2}	{ s_1 }



$\sum$ = { a, b }

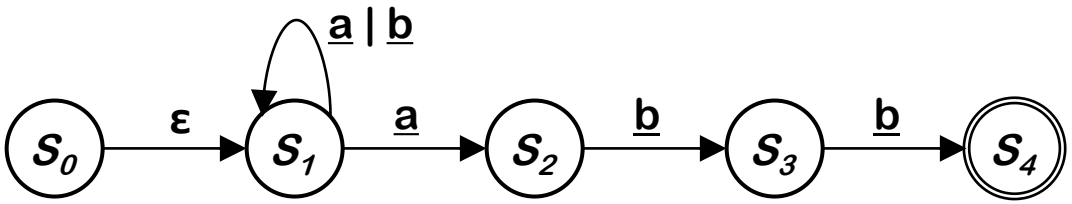
Dstates = { { s_0, s_1 }, { s_1, s_2 }, { s_1 } }

T = { s_1, s_2 }

α = a

```
Dstates ← {};
add ε-closure(s0) as an
      unmarked state to Dstates;
while ( there is an unmarked
    state T in Dstates ) {
  mark T;
  for each α ∈ ∑ {
       U ← ε-closure(Move(T, α))
       if ( U ∉ Dstates ) then
          add U as an unmarked
                       state to Dstates;
       δ [T, α] ← U
   }
}
```

δ	a	b
{s_0 ,s_1}	{s_1, s_2}	{ s_1 }



$\sum$ = { a, b }

Dstates = { { s_0, s_1 }, { s_1, s_2 }, { s_1 } }

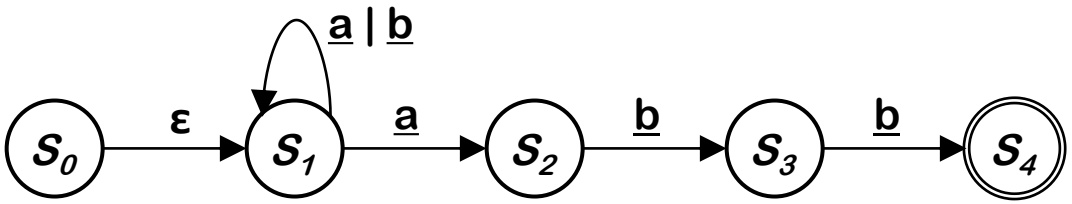
T = { s_1, s_2 }

α = a

```
Dstates ← {};
add ε-closure(s0) as an
      unmarked state to Dstates;
while ( there is an unmarked
    state T in Dstates ) {
  mark T;
  for each α ∈ ∑ {
       U ← ε-closure(Move(T, α))
       if ( U ∉ Dstates ) then
          add U as an unmarked
                       state to Dstates;
       δ [T, α] ← U
   }
}
```

U= { s_1 , s_2 }

δ	a	b
{s_0 ,s_1}	{s_1, s_2}	{ s_1 }



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

***Dstates* = { { s_0, s_1 }, { s_1, s_2 }, { s_1 } }**

***T* = { s_1, s_2 }**

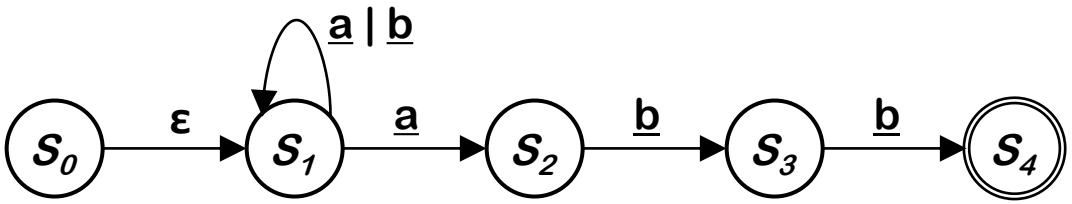
$\alpha = a$

$U = \{ s_1 , s_2 \}$

State U is already contained in Dstates (we are currently working on it!). Nothing to be done here...

```
Dstates ← {};
add ε-closure(s0) as an
      unmarked state to Dstates;
while ( there is an unmarked
     state T in Dstates ) {
  mark T;
  for each α ∈ ∑ {
       U ← ε-closure(Move(T, α))
       if ( U ∉ Dstates ) then
          add U as an unmarked
                        state to Dstates;
       δ [T, α] ← U
   }
}
```

δ	a	b
{s_0 ,s_1}	{s_1, s_2}	{ s_1 }



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

Dstates = { { s_0, s_1 }, { s_1, s_2 }, { s_1 } }

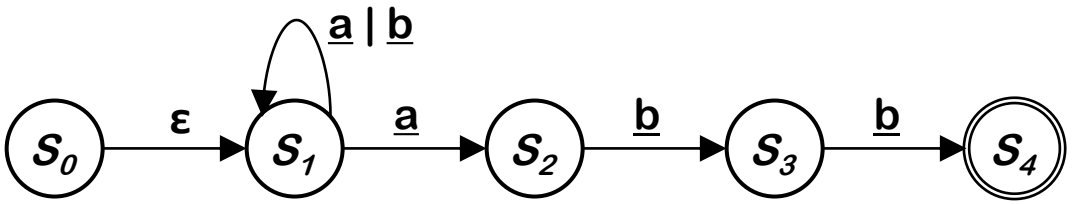
T = { s_1, s_2 }

α = a

U= { s_1 , s_2 }

```
Dstates ← {};
add ε-closure(s0) as an
      unmarked state to Dstates;
while ( there is an unmarked
     state T in Dstates ) {
  mark T;
  for each α ∈ ∑ {
       U ← ε-closure(Move(T, α))
       if ( U ∉ Dstates ) then
          add U as an unmarked
                        state to Dstates;
       δ [T, α] ← U
   }
}
```

δ	a	b
{s_0 ,s_1}	{s_1, s_2}	{ s_1 }
{ s_1, s_2 }	{ s_1, s_2 }	



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

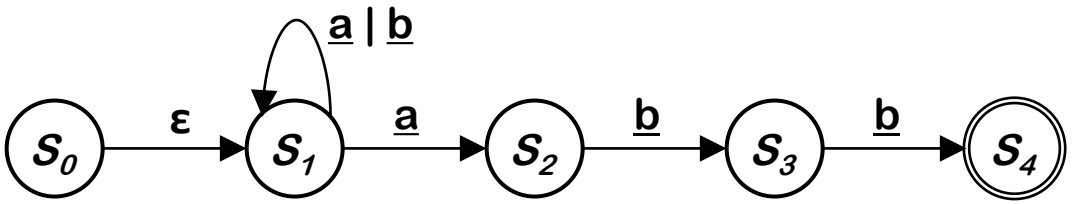
***Dstates* = { { s_0, s_1 }, { s_1, s_2 }, { s_1 } }**

***T* = { s_1, s_2 }**

$\alpha = b$

Dstates ← {};
add *ε-closure(s_0) as an unmarked state to Dstates;*
while *(there is an unmarked state T in Dstates) {*
  mark *T;*
  for each *α ∈ ∑ {*
    U ← ε-closure(Move(T, α))
    if *(U ∉ Dstates)* ***then***
      add *U as an unmarked state to Dstates;*
    δ [T, α] ← U
  }
}

δ	a	b
{s_0 ,s_1}	{s_1, s_2}	{ s_1 }
{ s_1, s_2 }	{ s_1, s_2 }	



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

Dstates = **{** $\{ s_0, s_1 \}, \{ s_1, s_2 \}, \{ s_1 \}$ **}**

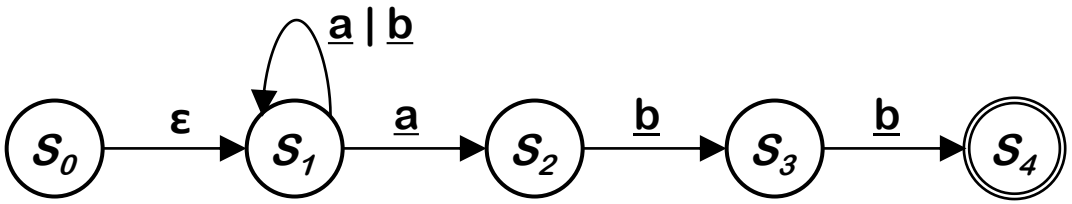
T = $\{ s_1, s_2 \}$

$\alpha = b$

Dstates ← {};
add *ε-closure*(s_0) *as an unmarked state to Dstates;*
while *(there is an unmarked state T in Dstates) {*
  mark *T;*
  for each $\alpha \in \sum$ {
    U ← *ε-closure(Move(T, α))*
    if *(U* ∉ *Dstates)* ***then***
      add *U as an unmarked state to Dstates;*
    δ [T, α] ← *U*
  }
}

$U = \{ s_1, s_3 \}$

δ	a	b
$\{s_0, s_1\}$	$\{s_1, s_2\}$	$\{ s_1 \}$
$\{ s_1, s_2 \}$	$\{ s_1, s_2 \}$	



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

***Dstates* = { { s_0, s_1 }, { s_1, s_2 }, { s_1 } }**

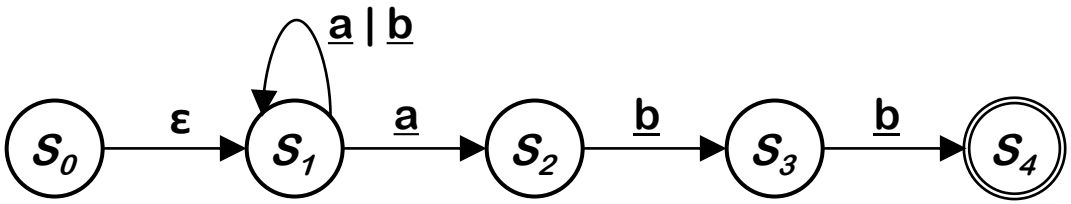
***T* = { s_1, s_2 }**

$\alpha = b$

Dstates ← {};
add *ε-closure(s_0) as an unmarked state to Dstates;*
while *(there is an unmarked state T in Dstates) {*
  mark *T;*
  for each *α ∈ ∑ {*
    U ← ε-closure(Move(T, α))
    if *(U ∉ Dstates)* ***then***
      add *U as an unmarked state to Dstates;*
    δ [T, α] ← U
  }
}

U = { s_1 , s_3 }

δ	a	b
{s_0 ,s_1}	{s_1, s_2}	{ s_1 }
{ s_1, s_2 }	{ s_1, s_2 }	



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

Dstates = { { s_0, s_1 }, { s_1, s_2 }, { s_1 }, { s_1 , s_3 } }

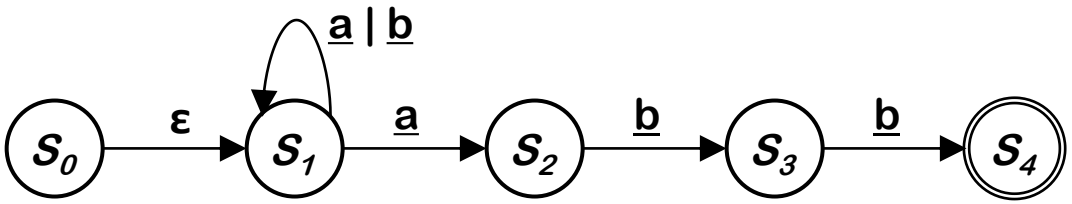
T = { s_1, s_2 }

α = b

Dstates ← {};
add *ε-closure*(s_0) *as an unmarked state to Dstates;*
while *(there is an unmarked state T in Dstates) {*
  mark *T;*
  for each $\alpha \in \sum$ {
    U ← *ε-closure(Move(T, α))*
    if *(U* ∉ *Dstates)* ***then***
      add *U as an unmarked state to Dstates;*
    δ [T, α] ← *U*
  }
}

U= { s_1 , s_3 }

δ	a	b
{s_0 ,s_1}	{s_1, s_2}	{ s_1 }
{ s_1, s_2 }	{ s_1, s_2 }	{ s_1 , s_3 }



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

Dstates = { { s_0, s_1 }, { s_1, s_2 }, { s_1 }, { s_1 , s_3 } }

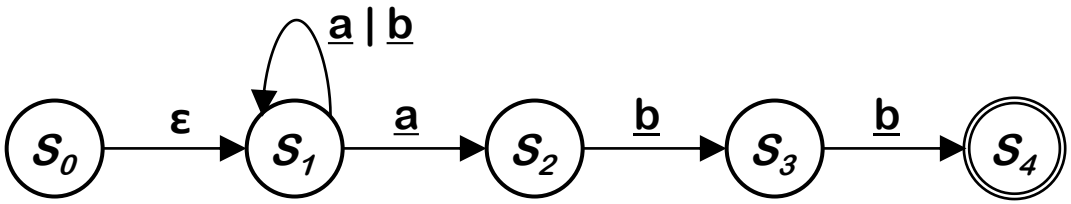
T = { s_1, s_2 }

α = b

```
Dstates ← {};
add ε-closure(s0) as an
      unmarked state to Dstates;
while ( there is an unmarked
     state T in Dstates ) {
  mark T;
  for each α ∈ ∑ {
       U ← ε-closure(Move(T, α))
       if ( U ∉ Dstates ) then
          add U as an unmarked
                       state to Dstates;
       δ [T, α] ← U
   }
}
```

{ s1 } is the next unmarked state in Dstates.

δ	a	b
{s_0 ,s_1}	{s_1, s_2}	{ s_1 }
{ s_1, s_2 }	{ s_1, s_2 }	{ s_1 , s_3 }



$\sum$ = { a, b }

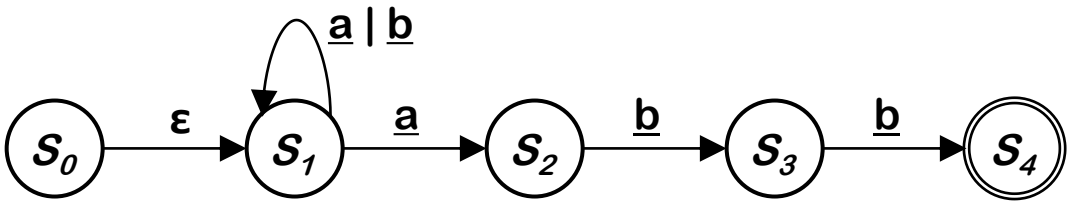
Dstates = { { s_0, s_1 }, { s_1, s_2 }, { s_1 }, { s_1 , s_3 } }

T = { s_1 }

α = a

```
Dstates ← {};
add ε-closure(s0) as an
      unmarked state to Dstates;
while ( there is an unmarked
    state T in Dstates ) {
  mark T;
  for each α ∈ ∑ {
       U ← ε-closure(Move(T, α))
       if ( U ∉ Dstates ) then
          add U as an unmarked
                       state to Dstates;
       δ [T, α] ← U
  }
}
```

δ	a	b
{s_0 ,s_1}	{s_1, s_2}	{ s_1 }
{ s_1, s_2 }	{ s_1, s_2 }	{ s_1 , s_3 }



$\sum$ = { a, b }

Dstates = { { s_0, s_1 }, { s_1, s_2 }, { s_1 }, { s_1 , s_3 } }

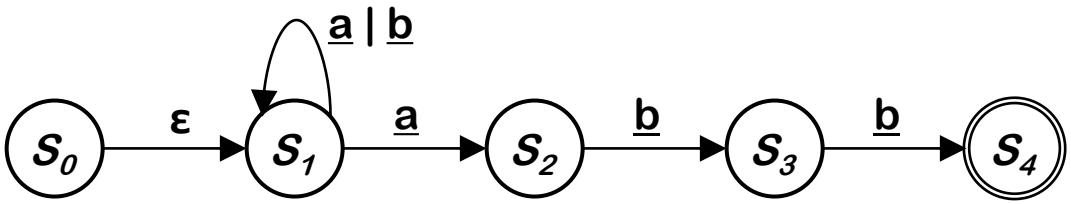
T = { s_1 }

α = a

```
Dstates ← {};
add ε-closure(s0) as an
      unmarked state to Dstates;
while ( there is an unmarked
    state T in Dstates ) {
  mark T;
  for each α ∈ ∑ {
       U ← ε-closure(Move(T, α))
       if ( U ∉ Dstates ) then
          add U as an unmarked
                       state to Dstates;
       δ [T, α] ← U
  }
}
```

U= { s_1 , s_2 }

δ	a	b
{s_0 ,s_1}	{s_1, s_2}	{ s_1 }
{ s_1, s_2 }	{ s_1, s_2 }	{ s_1 , s_3 }



$\sum$ = { a, b }

Dstates = { { s_0, s_1 }, { s_1, s_2 }, { s_1 }, { s_1 , s_3 } }

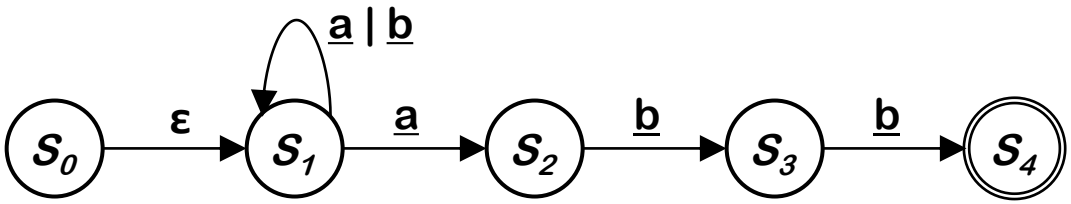
T = { s_1 }

α = a

```
Dstates ← {};
add ε-closure(s0) as an
      unmarked state to Dstates;
while ( there is an unmarked
    state T in Dstates ) {
  mark T;
  for each α ∈ ∑ {
       U ← ε-closure(Move(T, α))
       if ( U ∉ Dstates ) then
          add U as an unmarked
                       state to Dstates;
       δ [T, α] ← U
  }
}
```

U= { s_1 , s_2 }

δ	a	b
{s_0 ,s_1 }	{s_1, s_2 }	{ s_1 }
{ s_1, s_2 }	{ s_1, s_2 }	{ s_1 , s_3 }
{ s_1 }	{ s_1, s_2 }	



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

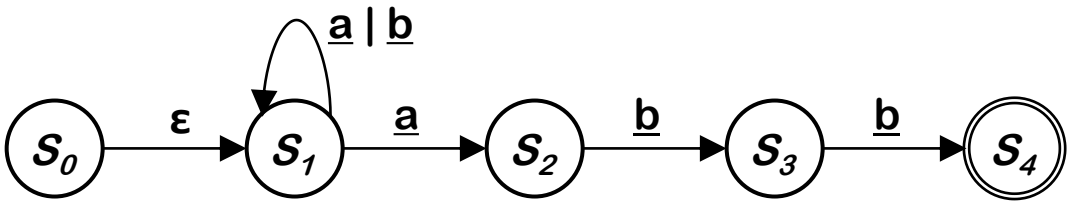
Dstates = **{** $\{ s_0, s_1 \}, \{ s_1, s_2 \}, \{ s_1 \}, \{ s_1 , s_3 \}$ **}**

T = $\{ s_1 \}$

$\alpha =$ *b*

```
Dstates ← {};
add ε-closure(s0) as an
      unmarked state to Dstates;
while ( there is an unmarked
     state T in Dstates ) {
  mark T;
  for each α ∈ ∑ {
       U ← ε-closure(Move(T, α))
       if ( U ∉ Dstates ) then
          add U as an unmarked
                        state to Dstates;
       δ [T, α] ← U
  }
}
```

δ	a	b
$\{s_0 , s_1 \}$	$\{s_1, s_2 \}$	$\{ s_1 \}$
$\{ s_1, s_2 \}$	$\{ s_1, s_2 \}$	$\{ s_1 , s_3 \}$
$\{ s_1 \}$	$\{ s_1, s_2 \}$	



$\sum$ = { a, b }

Dstates = { { s_0, s_1 }, { s_1, s_2 }, { s_1 }, { s_1 , s_3 } }

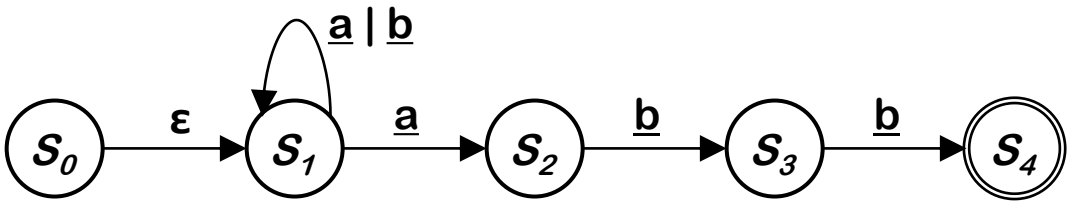
T = { s_1 }

α = b

```
Dstates ← {};
add ε-closure(s0) as an
      unmarked state to Dstates;
while ( there is an unmarked
    state T in Dstates ) {
  mark T;
  for each α ∈ ∑ {
       U ← ε-closure(Move(T, α))
       if ( U ∉ Dstates ) then
          add U as an unmarked
                       state to Dstates;
       δ [T, α] ← U
  }
}
```

U= { s_1 }

δ	a	b
{s_0 ,s_1 }	{s_1, s_2 }	{ s_1 }
{ s_1, s_2 }	{ s_1, s_2 }	{ s_1 , s_3 }
{ s_1 }	{ s_1, s_2 }	



$\sum$ = { a, b }

Dstates = { { s_0, s_1 }, { s_1, s_2 }, { s_1 }, { s_1 , s_3 } }

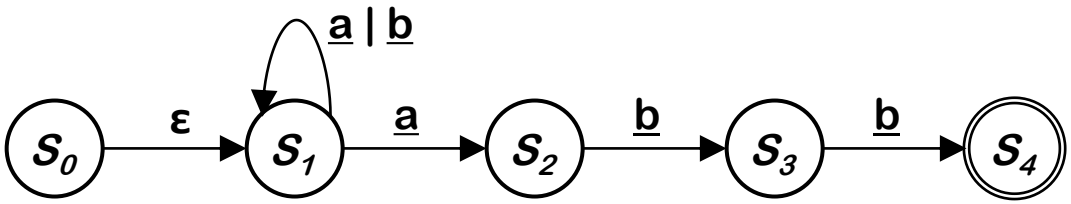
T = { s_1 }

α = b

```
Dstates ← {};
add ε-closure(s0) as an
    unmarked state to Dstates;
while ( there is an unmarked
    state T in Dstates ) {
  mark T;
  for each α ∈ ∑ {
      U ← ε-closure(Move(T, α))
      if ( U ∉ Dstates ) then
        add U as an unmarked
                  state to Dstates;
      δ [T, α] ← U
  }
}
```

U= { s_1 }

δ	a	b
{s_0 ,s_1 }	{s_1, s_2 }	{ s_1 }
{ s_1, s_2 }	{ s_1, s_2 }	{ s_1 , s_3 }
{ s_1 }	{ s_1, s_2 }	{ s_1 }



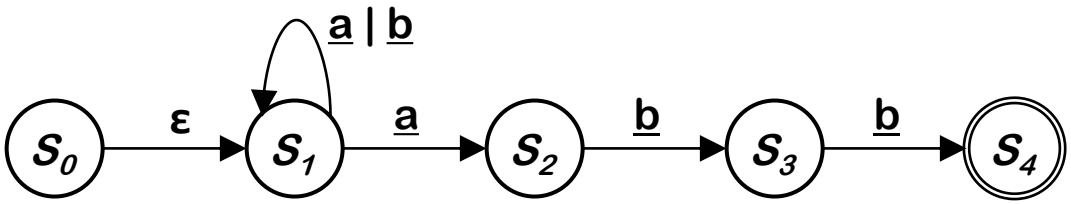
$\sum$ = { a, b }

Dstates = { { s_0, s_1 }, { s_1, s_2 }, { s_1 }, { s_1 , s_3 } }

α =

```
Dstates ← {};
add ε-closure(s0) as an
    unmarked state to Dstates;
while ( there is an unmarked
    state T in Dstates ) {
  mark T;
  for each α ∈ ∑ {
      U ← ε-closure(Move(T, α))
      if ( U ∉ Dstates ) then
        add U as an unmarked
                  state to Dstates;
      δ [T, α] ← U
  }
}
```

δ	a	b
{ s_0 , s_1 }	{ s_1, s_2 }	{ s_1 }
{ s_1, s_2 }	{ s_1, s_2 }	{ s_1 , s_3 }
{ s_1 }	{ s_1, s_2 }	{ s_1 }



$\sum$ = { a, b }

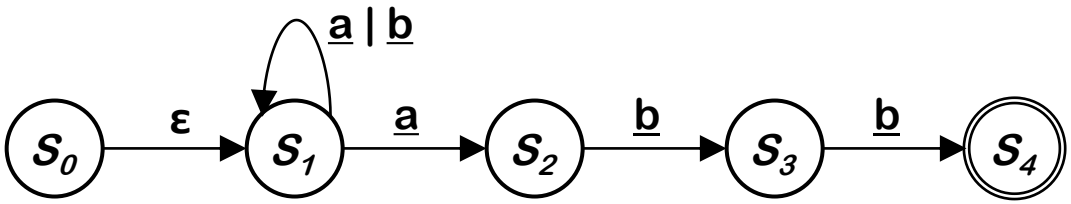
Dstates = { { s_0, s_1 }, { s_1, s_2 }, { s_1 }, { s_1 , s_3 } }

T = { s1 , s3 }

α =

```
Dstates ← {};
add ε-closure(s0) as an
      unmarked state to Dstates;
while ( there is an unmarked
     state T in Dstates ) {
  mark T;
  for each α ∈ ∑ {
       U ← ε-closure(Move(T, α))
       if ( U ∉ Dstates ) then
          add U as an unmarked
                        state to Dstates;
       δ [T, α] ← U
   }
}
```

δ	a	b
{s_0 ,s_1 }	{s_1, s_2 }	{ s_1 }
{ s_1, s_2 }	{ s_1, s_2 }	{ s_1 , s_3 }
{ s_1 }	{ s_1, s_2 }	{ s_1 }



$\sum$ = { a, b }

Dstates = { { s_0, s_1 }, { s_1, s_2 }, { s_1 }, { s_1 , s_3 } }

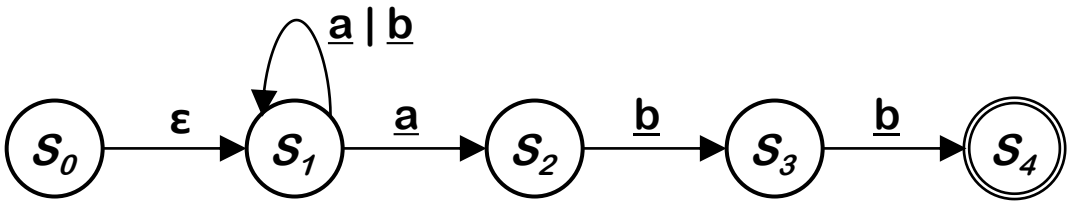
T = { s_1 , s_3 }

α = a

```
Dstates ← {};
add ε-closure(s0) as an
      unmarked state to Dstates;
while ( there is an unmarked
     state T in Dstates ) {
  mark T;
  for each α ∈ ∑ {
       U ← ε-closure(Move(T, α))
       if ( U ∉ Dstates ) then
          add U as an unmarked
                       state to Dstates;
       δ [T, α] ← U
  }
}
```

If we are in state T, reading an a, we get to { s_1, s_2 }. *This transition is added to* δ.

δ	a	b
{ s_0 , s_1 }	{ s_1, s_2 }	{ s_1 }
{ s_1, s_2 }	{ s_1, s_2 }	{ s_1 , s_3 }
{ s_1 }	{ s_1, s_2 }	{ s_1 }
{ s_1 , s_3 }	{ s_1, s_2 }	



$\sum$ = { a, b }

Dstates = { { s_0, s_1 }, { s_1, s_2 }, { s_1 }, { s_1 , s_3 } }

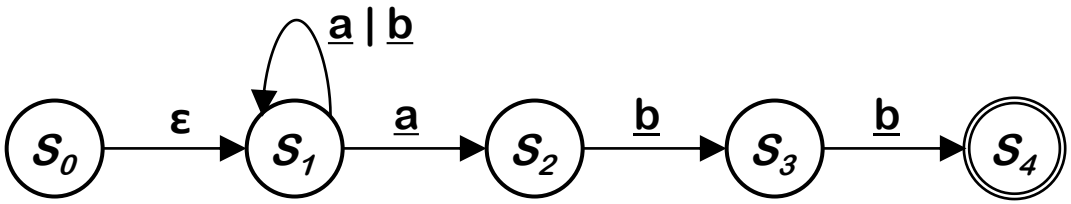
T = { s_1 , s_3 }

α = b

If we are in state T, reading a b, we get to { s_1, s_4 } ...

```
Dstates ← {};
add ε-closure(s0) as an
      unmarked state to Dstates;
while ( there is an unmarked
     state T in Dstates ) {
  mark T;
  for each α ∈ ∑ {
       U ← ε-closure(Move(T, α))
       if ( U ∉ Dstates ) then
          add U as an unmarked
                        state to Dstates;
       δ [T, α] ← U
  }
}
```

δ	a	b
{s_0 ,s_1 }	{s_1, s_2 }	{ s_1 }
{ s_1, s_2 }	{ s_1, s_2 }	{ s_1 , s_3 }
{ s_1 }	{ s_1, s_2 }	{ s_1 }
{ s_1 , s_3 }	{ s_1, s_2 }	



$\sum$ = { a, b }

Dstates = { { s_0, s_1 }, { s_1, s_2 }, { s_1 }, { s_1 , s_3 } }

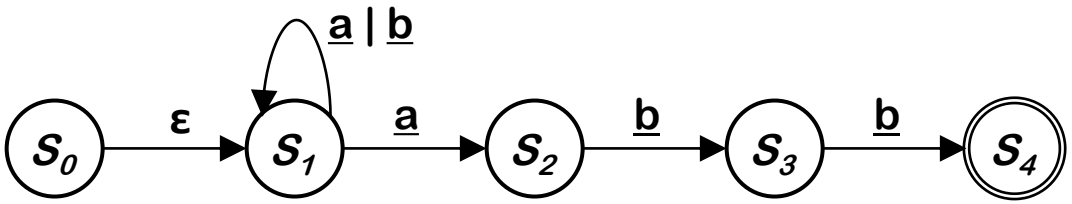
T = { s_1 , s_3 }

α = b

U= { s_1 , s_4 }

```
Dstates ← {};
add ε-closure(s0) as an
      unmarked state to Dstates;
while ( there is an unmarked
     state T in Dstates ) {
  mark T;
  for each α ∈ ∑ {
       U ← ε-closure(Move(T, α))
       if ( U ∉ Dstates ) then
          add U as an unmarked
                        state to Dstates;
       δ [T, α] ← U
   }
}
```

δ	a	b
{s_0 ,s_1 }	{s_1, s_2 }	{ s_1 }
{ s_1, s_2 }	{ s_1, s_2 }	{ s_1 , s_3 }
{ s_1 }	{ s_1, s_2 }	{ s_1 }
{ s_1 , s_3 }	{ s_1, s_2 }	



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

***Dstates* = { { s_0, s_1 }, { s_1, s_2 }, { s_1 }, { s_1 , s_3 } }**

***T* =** { s_1 , s_3 }

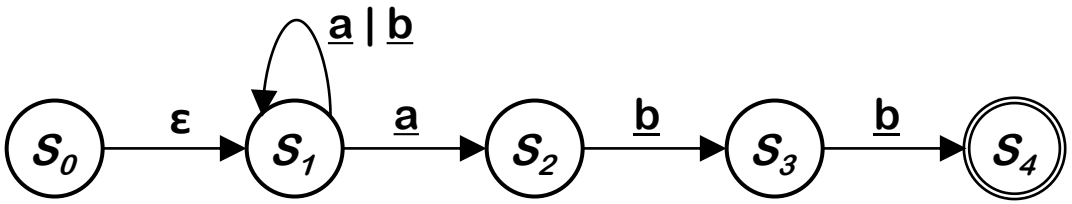
$\alpha = b$

U= { s_1 , s_4 }

U not yet in *Dstates* → add.

```
Dstates ← {};
add ε-closure(s0) as an
      unmarked state to Dstates;
while ( there is an unmarked
     state T in Dstates ) {
  mark T;
  for each α ∈ ∑ {
       U ← ε-closure(Move(T, α))
       if ( U ∉ Dstates ) then
          add U as an unmarked
                        state to Dstates;
       δ [T, α] ← U
   }
}
```

δ	a	b
{s_0 ,s_1 }	{s_1, s_2 }	{ s_1 }
{ s_1, s_2 }	{ s_1, s_2 }	{ s_1 , s_3 }
{ s_1 }	{ s_1, s_2 }	{ s_1 }
{ s_1 , s_3 }	{ s_1, s_2 }	



$\sum$ = { a, b }

Dstates = { { s_0, s_1 }, { s_1, s_2 }, { s_1 }, { s_1 , s_3 }, { s_1 , s_4 } }

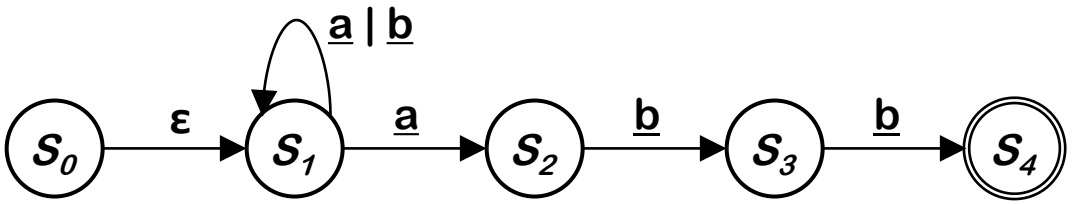
```
Dstates ← {};
add ε-closure(s0) as an
      unmarked state to Dstates;
while ( there is an unmarked
     state T in Dstates ) {
  mark T;
  for each α ∈ ∑ {
       U ← ε-closure(Move(T, α))
       if ( U ∉ Dstates ) then
          add U as an unmarked
                       state to Dstates;
       δ [T, α] ← U
  }
}
```

T = { s_1 , s_3 }

α = b

U= { s_1 , s_4 }

δ	a	b
{s_0 ,s_1 }	{s_1, s_2 }	{ s_1 }
{ s_1, s_2 }	{ s_1, s_2 }	{ s_1 , s_3 }
{ s_1 }	{ s_1, s_2 }	{ s_1 }
{ s_1 , s_3 }	{ s_1, s_2 }	{ s_1 , s_4 }



$\sum$ = { a, b }

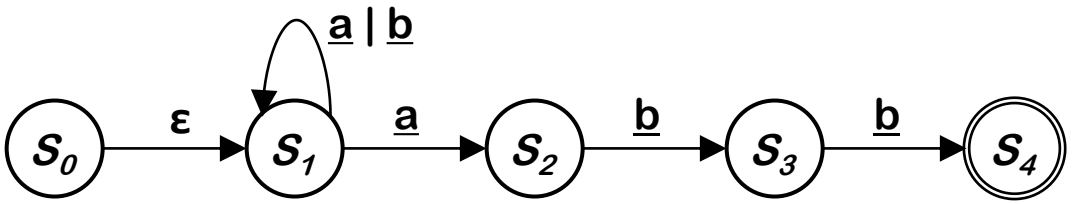
Dstates = { { s_0, s_1 }, { s_1, s_2 }, { s_1 }, { s_1 , s_3 }, { s_1 , s_4 } }

```
Dstates ← {};
add ε-closure(s0) as an
      unmarked state to Dstates;
while ( there is an unmarked
     state T in Dstates ) {
  mark T;
  for each α ∈ ∑ {
       U ← ε-closure(Move(T, α))
       if ( U ∉ Dstates ) then
          add U as an unmarked
                        state to Dstates;
       δ [T, α] ← U
  }
}
```

T={ s_1 , s_4 } is the last unmarked state.

Reading an a takes us to { s_1, s_2 }.
Reading a b takes us to { s_1 }.
Those two transitions are added to δ.

δ	a	b
{s_0 ,s_1 }	{s_1, s_2 }	{ s_1 }
{ s_1, s_2 }	{ s_1, s_2 }	{ s_1 , s_3 }
{ s_1 }	{ s_1, s_2 }	{ s_1 }
{ s_1 , s_3 }	{ s_1, s_2 }	{ s_1 , s_4 }
{ s_1 , s_4 }	{ s_1, s_2 }	{ s_1 }



$\sum$ = { a, b }

Dstates = { { s_0, s_1 }, { s_1, s_2 }, { s_1 }, { s_1 , s_3 }, { s_1 , s_4 } }

```
Dstates ← {};
add ε-closure(s0) as an
      unmarked state to Dstates;
while ( there is an unmarked
     state T in Dstates ) {
  mark T;
  for each α ∈ ∑ {
       U ← ε-closure(Move(T, α))
       if ( U ∉ Dstates ) then
          add U as an unmarked
                       state to Dstates;
       δ [T, α] ← U
  }
}
```

No more unmarked states in Dstates.

The algorithm stops.

δ	a	b
{ s_0 , s_1 }	{ s_1, s_2 }	{ s_1 }
{ s_1, s_2 }	{ s_1, s_2 }	{ s_1 , s_3 }
{ s_1 }	{ s_1, s_2 }	{ s_1 }
{ s_1 , s_3 }	{ s_1, s_2 }	{ s_1 , s_4 }
{ s_1 , s_4 }	{ s_1, s_2 }	{ s_1 }

δ	a	b
$\{S_0, S_1\}$	$\{S_1, S_2\}$	$\{S_1\}$
$\{S_1, S_2\}$	$\{S_1, S_2\}$	$\{S_1, S_3\}$
$\{S_1\}$	$\{S_1, S_2\}$	$\{S_1\}$
$\{S_1, S_3\}$	$\{S_1, S_2\}$	$\{S_1, S_4\}$
$\{S_1, S_4\}$	$\{S_1, S_2\}$	$\{S_1\}$

All DFA states that contain an accepting NFA state become accepting states in the DFA!

The DFA for the above transition function δ: