Finite Automata

- Finite automata are the simplest machines to recognize patterns
- They are essentially graphs like transition diagrams. They simply say "yes" or "no" about each possible input string.
  - **Nondeterministic finite automata (NFA, 非确定有穷自动机):** A symbol can label several edges out of the same state (allowing multiple target states), and the empty string ϵ is a possible label.
  - **Deterministic finite automata (DFA, 确定有穷自动机):** For each state and for each symbol in the input alphabet, there is exactly one edge with that symbol leaving that state.
- NFA and DFA recognize the same languages, **regular languages**, which regexps can describe.

NFA与DFA都识别正则语言.

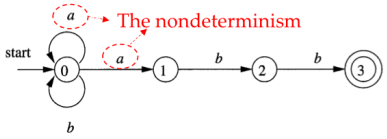
Nondeterministic Finite Automata (NFA)

- An **NFA** is a 5-tuple, consisting of:
  1. A finite set of states S
  2. A set of input symbols Σ, the *input alphabet*. We assume that the empty string ϵ is never a member of Σ
  3. A *transition function* that gives, for each state, and for each symbol in $\Sigma \cup \{\epsilon\}$ a set of *next states*
  4. A *start state* (or initial state) s_0 from S
  5. A set of *accepting states* (or *final states*) F, a subset of S

Example

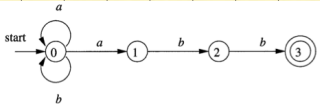
- S = {0, 1, 2, 3}
- Σ = {a, b}
- Start state: 0
- Accepting states: {3}
- Transition function
  - (0, a) → {0, 1} (0, b) → {0}
  - (1, b) → {2} (2, b) → {3}

The NFA can be represented as a Transition Graph:



Transition Table

Transition Table

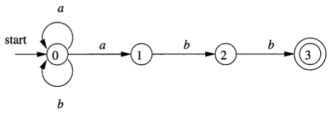


- Another representation of an NFA
  - Rows correspond to states
  - Columns correspond to the input symbols or ϵ
  - The table entry for a state-input pair lists the set of next states
  - ∅: the transition function has no info about the state-input pair

STATE	a	b	ϵ
0	{0, 1}	{0}	∅
1	∅	{2}	∅
2	∅	{3}	∅
3	∅	∅	∅

Acceptance of Input Strings

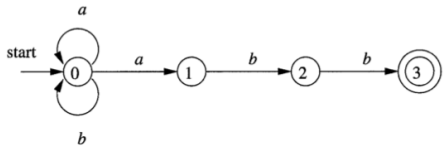
- An NFA accepts an input string x if and only if
  - There is a path in the transition graph from the start state to one accepting state, such that the symbols along the path form x (ϵ labels are ignored).



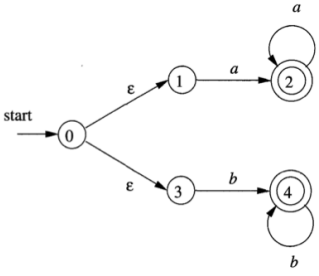
accepts the string *aabb*

- The language defined or accepted by an NFA
  - The set of strings labelling some path from the start state to an accepting state

NFA and Regular Language



$L((a|b)^*abb)$

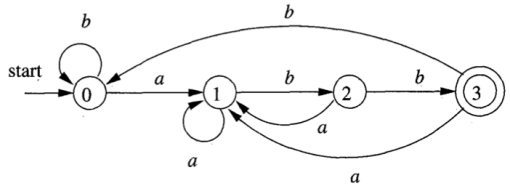


$L(aa^*|bb^*)$

Deterministic Finite Automata (DFA)

- A **DFA** is a special NFA where:
  - There are no moves on input ϵ
  - For each state s and input symbol a, there is exactly one edge out of s labeled a (i.e., exactly one target state)

不会对ε有移动
对于某个输入，下一状态是确定的.



Simulating a DFA

- **Input:**
  - String x terminated by an end-of-file character **eof**.
  - DFA D with *start state* s_0, *accepting states* F, and transition function *move*
- **Output:** Answer "yes" if D accepts x; "no" otherwise

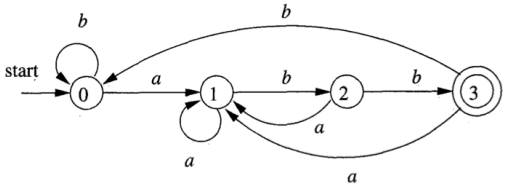
```
s = s0;
c = nextChar();
while ( c != eof ) {
        s = move(s, c);
        c = nextChar();
}
if ( s is in F ) return "yes";
else return "no";
```

We can see from the algorithm:

- DFA can efficiently accept/reject strings (i.e., recognize patterns)

Example

- Given the input string *ababb*, the DFA below enters the sequence of states 0, 1, 2, 1, 2, 3 and returns "yes"



What's the language defined by this DFA?

From Regular Expressions to Automata

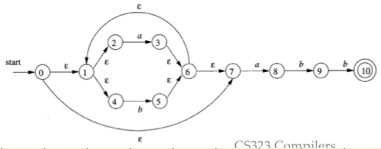
- Regexps concisely & precisely describe the patterns of tokens
- DFA can efficiently recognize patterns (comparatively, the simulation of NFA is less straightforward*)
- When implementing lexical analyzers, regexps are often converted to DFA:
  - Regexp → NFA → DFA
  - Algorithms: Thompson's construction + subset construction

* There may be multiple transitions at a state when seeing a symbol

Regexp —Thompson→ NFA —Subset→ DFA

Conversion of an NFA to a DFA

- The **subset construction** algorithm (子集构造法)
  - **Insight:** Each state of the constructed DFA corresponds to a set of NFA states
    - Why? Because after reading the input $a_1a_2\ldots a_n$, the DFA reaches one state while the NFA may reach multiple states
  - **Basic idea:** The algorithm simulates "in parallel" all possible moves an NFA can make on a given input string to map a set of NFA states to a DFA state.



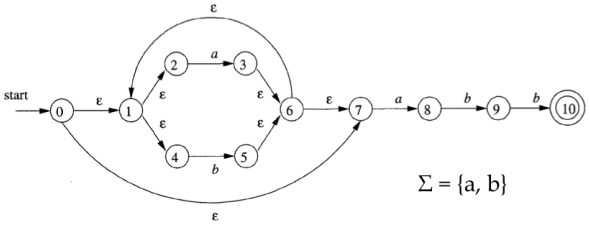
After reading "a", the NFA may reach any of these states:

3, 6, 1, 7, 2, 4, 8

DFA的每个状态都对应多个NFA的状态

Example for Algorithm Illustration

- The NFA below accepts the string *babb*
  - There exists a path from the start state 0 to the accepting state 10, on which the labels on the edges form the string *babb*



Σ = {a, b}

Subset Construction Technique

- Operations used in the algorithm:
  - **ϵ-closure(s):** Set of NFA states reachable from NFA state s on ϵ-transitions alone
  - **ϵ-closure(T):** Set of NFA states reachable from some NFA state s in set T on ϵ-transitions alone
    - That is, $\bigcup_{s \text{ in } T} \epsilon\text{-closure}(s)$
  - ***move*(T, a):** Set of NFA states to which there is a transition on input symbol a from some state s in T (i.e., the target states of those states in T when seeing a)

最长的 ε 转移路径.

针对其中几个状态

- **Computing ϵ-*closure*(T)**
  - It is a graph traversal process (only consider ϵ edges)
  - Computing ϵ-*closure*(s) is the same (when T has only one state)

```
push all states of T onto stack;
initialize ε-closure(T) to T;
while ( stack is not empty ) {
        pop t, the top element, off stack;
        for ( each state u with an edge from t to u labeled ε )
                if ( u is not in ε-closure(T) ) {
                        add u to ε-closure(T);
                        push u onto stack;
                }
}
```

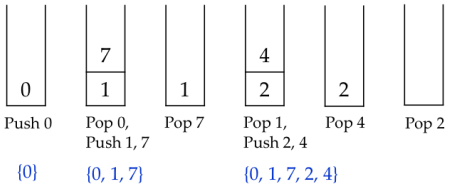
图搜索 ε 边.

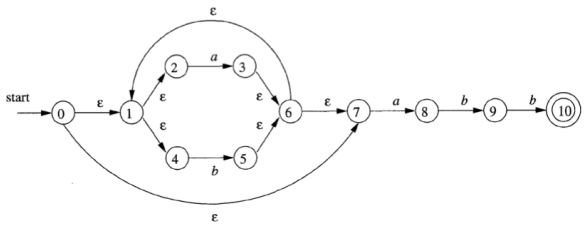
$T \in \epsilon\text{-closure}(T)$

Example

- ϵ-*closure*(0) = ?

  {0,1,7,2,4}



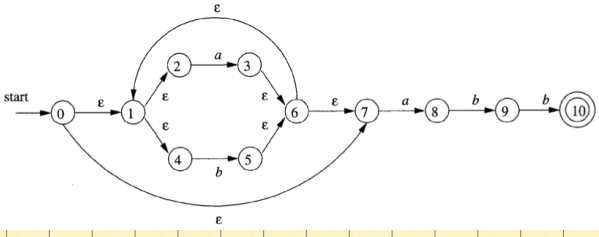


- ϵ-*closure*({3, 8}) = ?

ε-closure(3) = {3, 6, 7, 1, 2, 4}

ε-closure(8) = {8}

⇒ ε-closure({3, 8}) = {3, 6, 7, 1, 2, 4, 8}



- The construction of the DFA *D*'s states, *Dstates*, and the transition function *Dtran* is also a search process
  - Initially, the only state in *Dstates* is ϵ-*closure*(s_0) and it is unmarked
    - Unmarked state means that its next states have not been explored

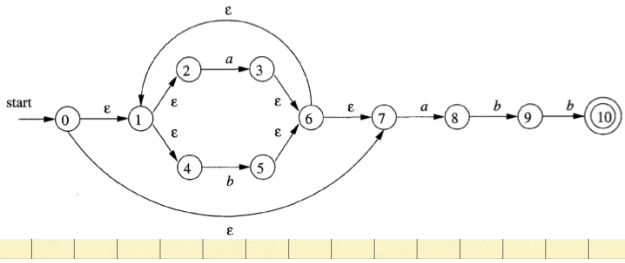
```
while ( there is an unmarked state T in Dstates ) {
    mark T;
    for ( each input symbol a ) {  // find the next states of T
        U = ε-closure(move(T, a));
        if ( U is not in Dstates )
            add U as an unmarked state to Dstates;
        Dtran[T, a] = U;
    }
}
```

DFA的初始状态是ε-closure(1)

需要遍历字母表的每个元素.

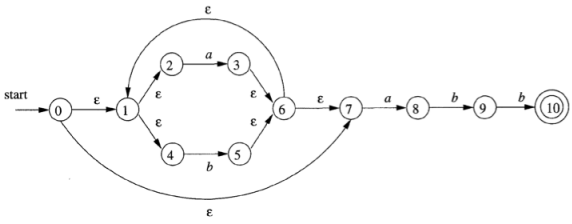
Example

- Initially, Dstates only has one unmarked state:
  - ϵ-*closure*(0) = {0, 1, 2, 4, 7} -- A
- Dtran is empty



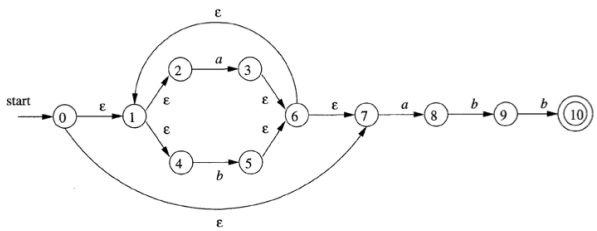
ϵ-*closure*(*move*[A, a])
= ϵ-*closure*({3, 8})
= {1, 2, 3, 4, 6, 7, 8}

- We get an unseen state {1, 2, 3, 4, 6, 7, 8} -- B
- Update Dstates: {A, B}
- Update Dtran: {[A, a] ➔ B}



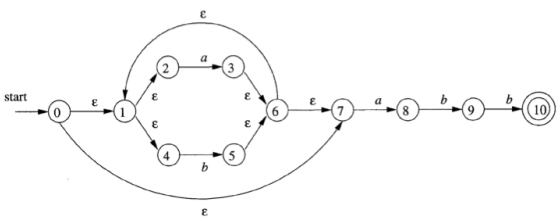
ϵ-*closure*(*move*[A, b])
= ϵ-*closure*({5})
= {1, 2, 4, 5, 6, 7}

- We get an unseen state {1, 2, 4, 5, 6, 7} -- C
- Update Dstates: {A, B, C}
- Update Dtran: {[A, a] ➔ B, [A, b] ➔ C}



ϵ-*closure*(*move*[B, a])
= ϵ-*closure*({3, 8})
= {1, 2, 3, 4, 6, 7, 8}

- The state {1, 2, 3, 4, 6, 7, 8} already exists (B)
- No need to update Dstates: {A, B, C}
- Update Dtran: {[A, a] ➔ B, [A, b] ➔ C, [B, a] ➔ B}

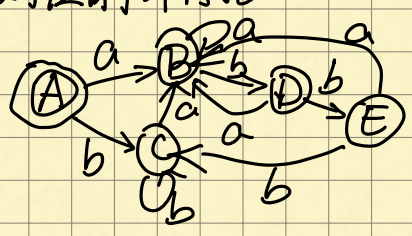


- Eventually, we will get the following DFA:
  - Start state: A; Accepting states: {E}

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

This DFA can be further minimized: A and C have the same moves on all symbols and can be merged.

对应的DFA为



Regular Expression to NFA

Thompson's construction algorithm (Thompson构造法)

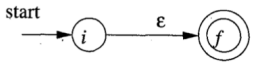
- The algorithm works recursively by splitting a regular expression into subexpressions, from which the NFA will be constructed using the following rules:
  - **Two basis rules (基本规则):** handle subexpressions with no operators
  - **Three inductive rules (归纳规则):** construct larger NFA's from the smaller NFA's for subexpressions

递归将正则表达式分割成子表达式

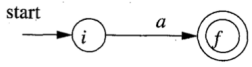
Thompson's Construction Algorithm

Two basis rules:

1. The empty expression ϵ is converted to

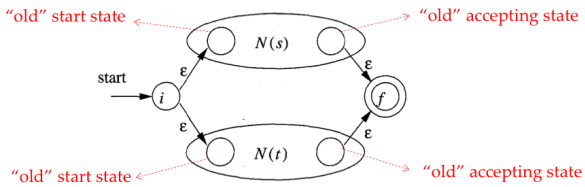


2. Any subexpression a (a single symbol in input alphabet) is converted to



The inductive rules: the union case

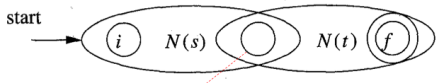
- $s \mid t$: $N(s)$ and $N(t)$ are NFA's for subexpressions s and t



By construction, the NFA's have only one start state and one accepting state

The inductive rules: the concatenation case

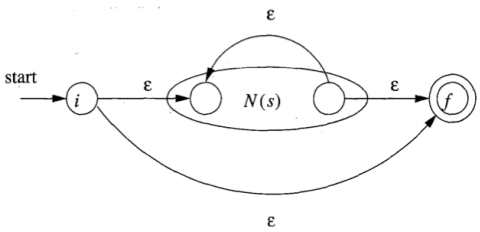
- st : $N(s)$ and $N(t)$ are NFA's for subexpressions s and t



Merging the accepting state of $N(s)$ and the start state of $N(t)$

The inductive rules: the Kleene star case

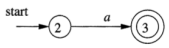
- s^* : $N(s)$ is the NFA for subexpression s



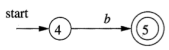
Example

Use Thompson's algorithm to construct an NFA for the regexp r = (a|b)*a

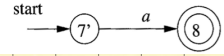
1. NFA for the first **a** (apply basis rule #1)



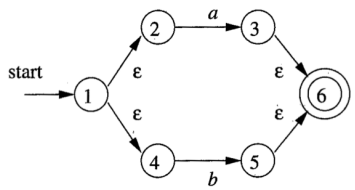
2. NFA for the first **b** (apply basis rule #1)



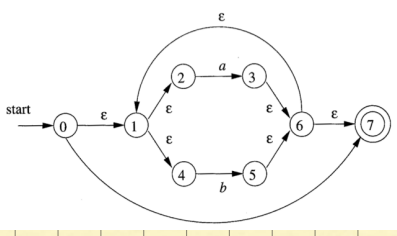
5. NFA for the second **a** (apply basic rule #1)



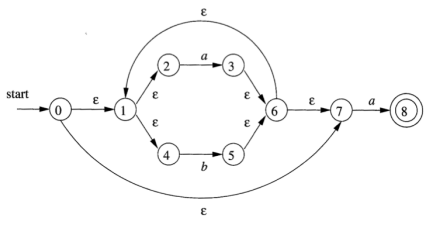
3. NFA for **(a|b)** (apply inductive rule #1)



4. NFA for **(a|b)*** (apply inductive rule #3)

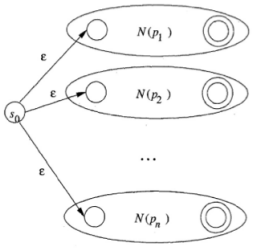


6. NFA for **(a|b)*a** (apply inductive rule #2)



Combining NFA's

- **Why?** In the lexical analyzer, we need a single automaton to recognize lexemes matching any pattern (in the `lex` program)
- **How?** Introduce a new start state with ϵ-transitions to each of the start states of the NFA's for pattern p_i



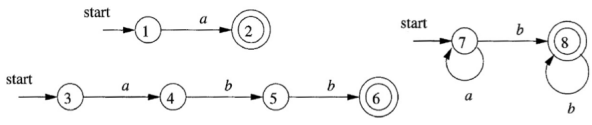
- The language that can be accepted by the big NFA is the union of the languages that can be accepted by the small NFA's
- Different accepting states correspond to different patterns

DFA's for Lexical Analyzers

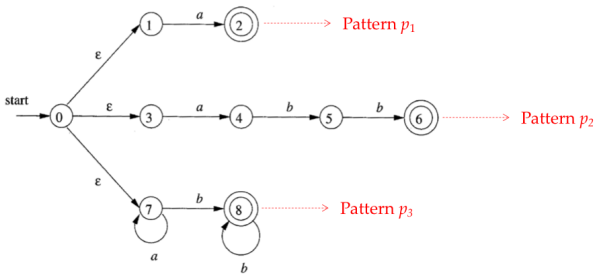
- Convert the NFA for all the patterns into an equivalent DFA, using the subset construction algorithm
- An accepting state of the DFA corresponds to a subset of the NFA states, in which at least one is an accepting NFA state
  - If there are more than one accepting NFA state, this means that conflicts arise (the prefix of the input string matches multiple patterns)
  - Upon conflicts, find the first pattern whose accepting state is in the set and make that pattern the output of the DFA state

Example

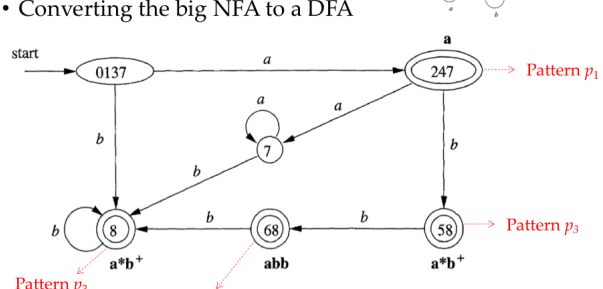
- Suppose we have three patterns: p_1, p_2, and p_3
  - **a** {action A_1 for pattern p_1}
  - **abb** {action A_2 for pattern p_2}
  - **a*b\+** {action A_3 for pattern p_3}
- We first construct an NFA for each pattern



- Combining the three NFA's



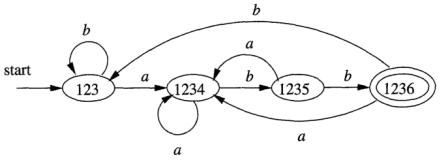
- Converting the big NFA to a DFA



当有状态冲突时，可手动指定pattern的优先级

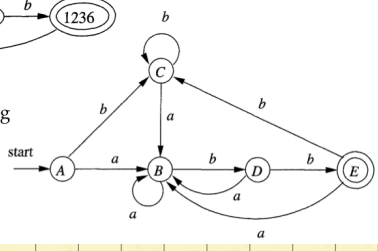
Minimizing # States of a DFA

- There can be many DFA's recognizing the same language



Two equivalent DFA's, both recognizing regular language $L((a|b)^*abb)$

*Self-study materials



States A and C are equivalent:
Neither is accepting, and on any symbol they transfer to the same state

A and C behave like 123

- There is always a unique minimum-state DFA for any regular language (state name does not matter)

对每个正则表达式，都有唯一的最小状态DFA.

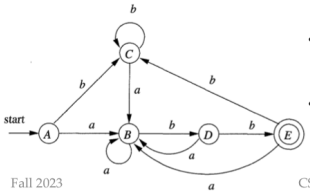
- The minimum-state DFA can be constructed from any DFA for the same language by grouping sets of equivalent states

Distinguishing States

- **Distinguishable states**
  - We say that string x distinguishes state s from state t if exactly one of the states reached from s and t by following the path with label x is an accepting state
  - States s and t are distinguishable if there exists some string that distinguishes them

字符串x能区分状态s和t，当从状态s或t开始，沿着x指定的路径，能到达accept状态.

- **For two indistinguishable states,** scanning any string will lead to the same state. Such states are equivalent and should be merged.



- The empty string ϵ distinguishes any accepting state from any nonaccepting state
- The string bb distinguishes state A from B, since bb takes A to a nonaccepting state C, but takes B to an accepting state E

DFA State-Minimization Algorithm

Works by partitioning the states of a DFA into groups of states that cannot be distinguished (an iterative process)

将不可区分的状态合为一组

- The algorithm maintains a partition (划分), whose groups are sets of states that have not yet been distinguished
- Any two states from different groups are known to be distinguishable
- When the partition cannot be refined further by breaking any group into smaller groups, we have the minimum-state DFA

The Partitioning Part

1. Start with an initial partition $\prod$ with two groups, F and $S - F$, the accepting and nonaccepting states of D
2. Apply the procedure below to construct a new partition $\prod_{new}$

```
initially, let Π_new = Π;
for ( each group G of Π ) {
        partition G into subgroups such that two states s and t
                are in the same subgroup if and only if for all
                input symbols a, states s and t have transitions on a
                to states in the same group of Π;
        /* at worst, a state will be in a subgroup by itself */
        replace G in Π_new by the set of all subgroups formed;
}
```

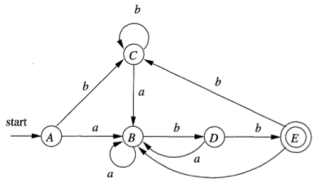
3. If $\prod_{new} == \prod$, let $\prod_{final} = \prod$ and the partitioning finishes; Otherwise, $\prod = \prod_{new}$ and repeat step 2

The Construction Part

- Choose one state in each group of $\prod_{\text{final}}$ as the *representative* for that group. The representatives will be the states of the minimum-state DFA D'
  - The start state of D' is the representative of the group containing the start state of D
  - The accepting states of D' are the representatives of those groups that contain an accepting state of D
  - Establish transitions:
    - Let s be the representative of group G in $\prod_{\text{final}}$; Let the transition of D from s on input a be to state t; Let r be the representative of t's group H
    - Then in D', there is a transition from s to r on input a

Example

- Initial partition: {*A*, *B*, *C*, *D*}, {*E*}
- Handling group {*A*, *B*, *C*, *D*}: *b* splits it to two subgroups {*A*, *B*, *C*} and {*D*}
- Handling group {*A*, *B*, *C*}: *b* splits it to two subgroups {*A*, *C*} and {*B*}
- Picking A, B, D, E as representatives to construct the minimum-state DFA



STATE	*a*	*b*
A	*B*	*A*
B	*B*	*D*
D	*B*	*E*
E	*B*	*A*

State Minimization in Lexical Analyzers

- The basic idea is the same as the state-minimization algorithm for DFA.
- Differences are:
  - Each accepting state in the lexical analyzer's DFA corresponds to a different pattern. These states are not equivalent.
  - So, the initial partition should be: one group of non-accepting states + groups of accepting states for different patterns

Example

- Initial partition: {0137, 7}, {247}, {68}, {8, 58}, {∅}
  - We add a dead state ∅: we suppose has transitions to itself on inputs *a* and *b*. It is also the target of missing transitions on *a* from states 8, 58, and 68.