CSC 448: Compiler Design

Lecture 2

Joseph Phillips
De Paul University

2016 April 6

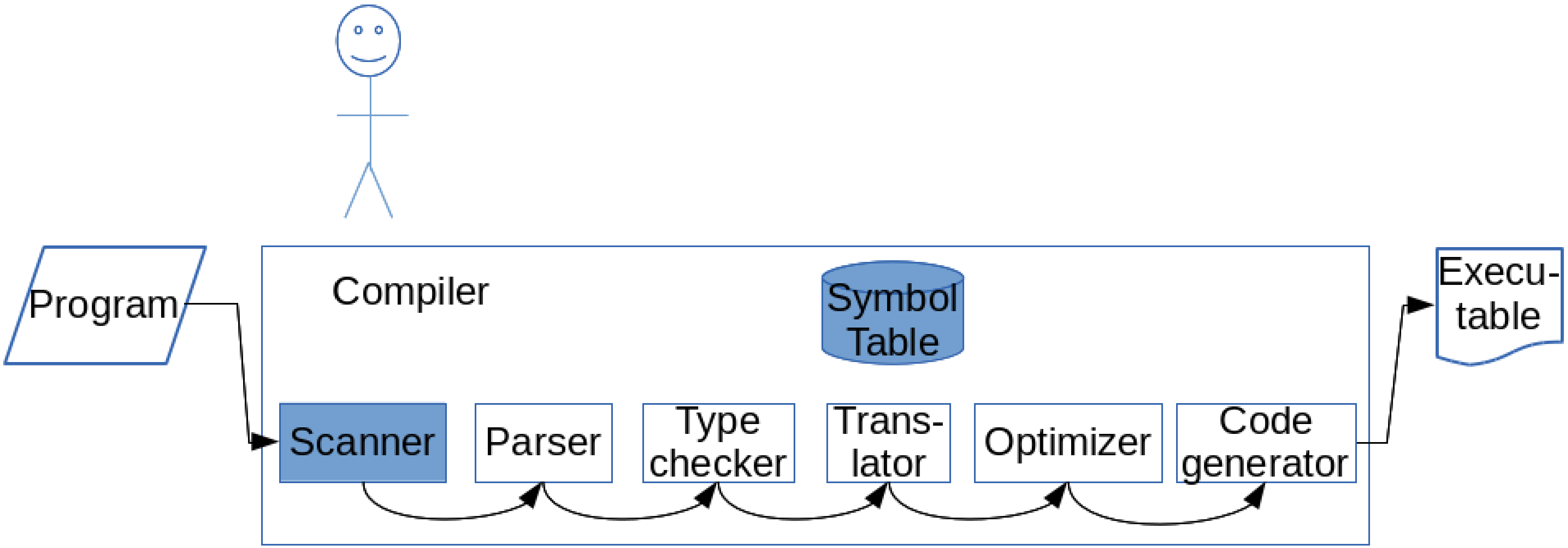
Reading

- Charles Fischer, Ron Cytron, Richard LeBlanc Jr. “Crafting a Compiler” Addison-Wesley. 2010.
  - Chapter 3: Scanning – Theory and Practice

Topics:

- Scanning (Theory)

Overview:



Program
Compiler
Symbol Table
Scanner
Parser
Type checker
Trans-lator
Optimizer
Code generator
Execu-table

What could be so difficult about deciding where one lexeme ends and the next begins?

- Escape characters:
  - `"What is this ->\" character?"`
- Interactions among characters:
  - Does your language accept ranges like `0..9` ?
  - If it does, then it is harder to recognize “partial floats” like `0.` and `.9` .

Regular Expressions

- Allow us to specify what lexemes should look like (“*declarative programming*”).
- Then we can give these definitions to a **scanner-generator** to make code for us.
  - Generates *procedural code* in C, Java, *etc*. that actually recognizes it

Regular Expressions

- Vocabulary (Σ) and its subset:
  - chars actually used in language
  - `D` = `(0|1|2|3|4|5|6|7|8|9)` = `(0|...|9)` = the set of digit chars
  - `UC` = `(A|...|Z)` = the set of uppercase chars
  - `LC` = `(a|...|z)` = the set of lowercase chars
- λ: empty string
- Meta-chars: chars that build expressions
  - `( ) | + *`

Regular Expressions

- Expressions:
  - Alternation: ***(this|that)*** = “this or that”
    - `0 | 1 = { "0", "1" }`
  - Concatenation: ***this that*** = “this, followed by that”
    - **`01 = { "01" }`**
    - **`(0|1) (0|1) = { "00", "01",  "10", "11" }`**
  - Kleene closure: **`this`*** = “zero or more concatenated this”
    - **`(0|1)`*** = the set of binary numbers ***and*** λ
  - Positive closure: **`this`**$^+$ = “one or more concatenated this”
    - **`(0|1)`**$^+$ = the set of binary numbers ***not including*** λ
  - Positive closure: **`this`**^{k} = “precisely k-concatenated this-es”
    - **`(0|1)`**^{2} = **`{ "00", "01",  "10", "11" }`**
  - Negation: **`not(these)`** = all the characters or strings not in these
  -

Regular Expression Examples:

- Common identifiers:
  - UC = `(A|...|Z)` = uppercase chars
  - LC = `(a|...|z)` = lowercase chars
  - D = `(0|...|9)` = digit
  - $(\mathtt{UC}|\mathtt{LC}|\_)\ (\mathtt{UC}|\mathtt{LC}|\_|\mathtt{D})^*$
- Integers:
  - $(+|-|\lambda)\ \mathtt{D}^+$

Your turn!

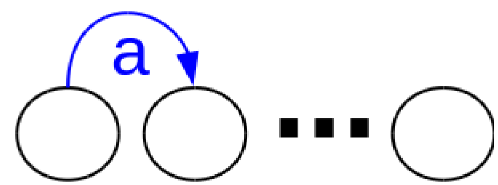
- Write regular expressions for:
  - Floating point numbers (with and without scientific notation)
  - C string constants (where \ is used as an escape character before ", n, t, and \).

Deterministic Finite Automata

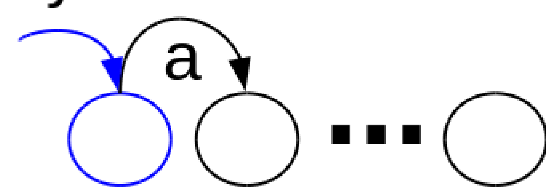
A ***finite*** set of states:

A ***finite*** char vocabulary: $\Sigma = \{ a, b \}$

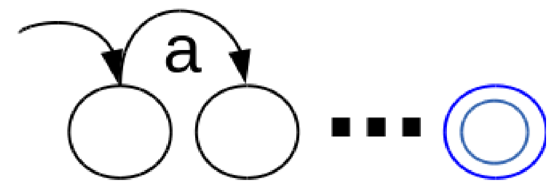
A ***finite*** and ***deterministic set of transitions*** between states labeled by vocabulary



A ***unique*** starting state:

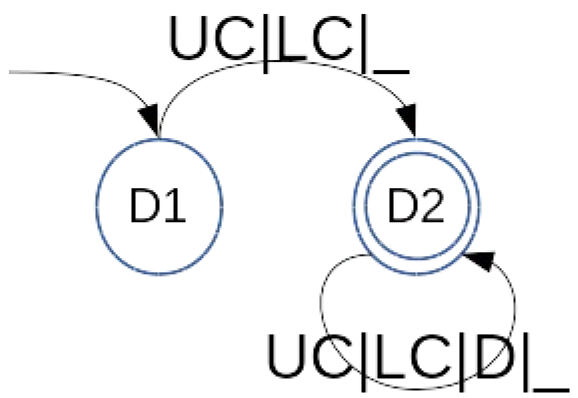


A set of ***accepting*** or ***final*** states:

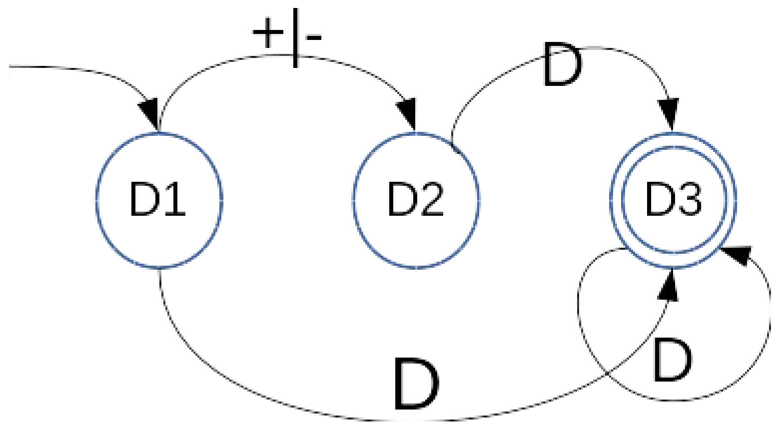


DFA Examples

DFA for identifiers



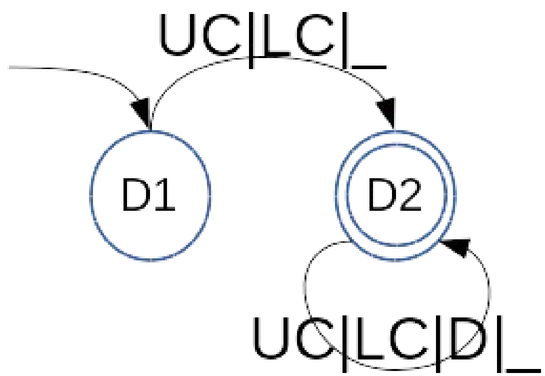
DFA for integers



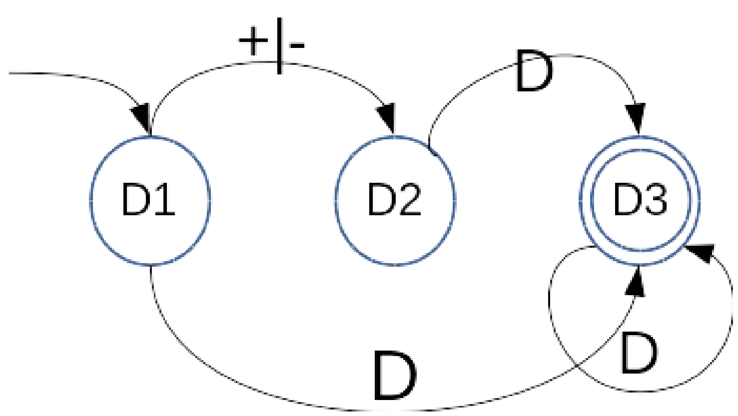
Q: Why get a DFA from a Regular Expression?

A: Because a DFA could easily be turned into table. Then, a standard algorithm can use table to recognize lexemes.

Example



	Is accepting?	UC	LC	_	D
D1	No	D2	D2	D2	--
D2	Yes	D2	D2	D2	D2



	Is accepting?	+	-	D
D1	No	D2	D2	D3
D2	No	--	--	D3
D3	Yes	--	--	D3

Your Turn!

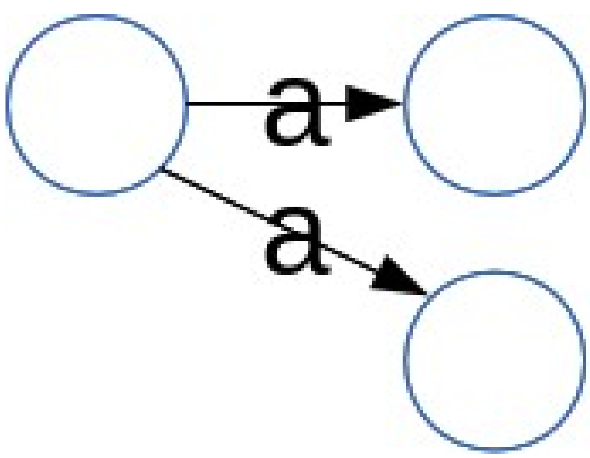
- Write a DFA for:
  - Floating point numbers (with and without scientific notation)
  - C string constants (where \ is used as an escape character before ", n, t, and \).

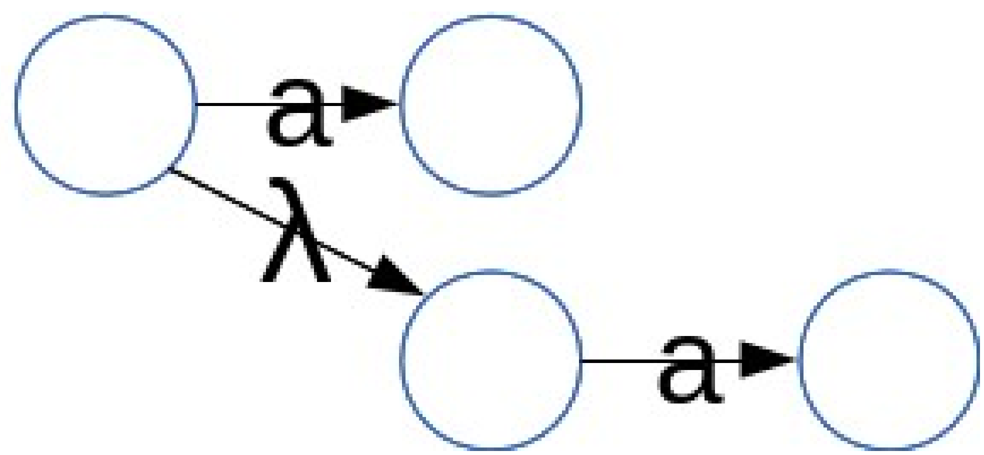
So, I can specify a regular expression and get software to recognize my lexemes?

- Yep:
  1) Regular expression, to
  2) ***Non-deterministic*** finite state machine, to
     - Allows λ transitions
     - Therefore, machine can be in multiple states at same time
  3) ***Deterministic*** finite state machine, to
  4) Recognizing table

Regular Expression -> NFA (1)

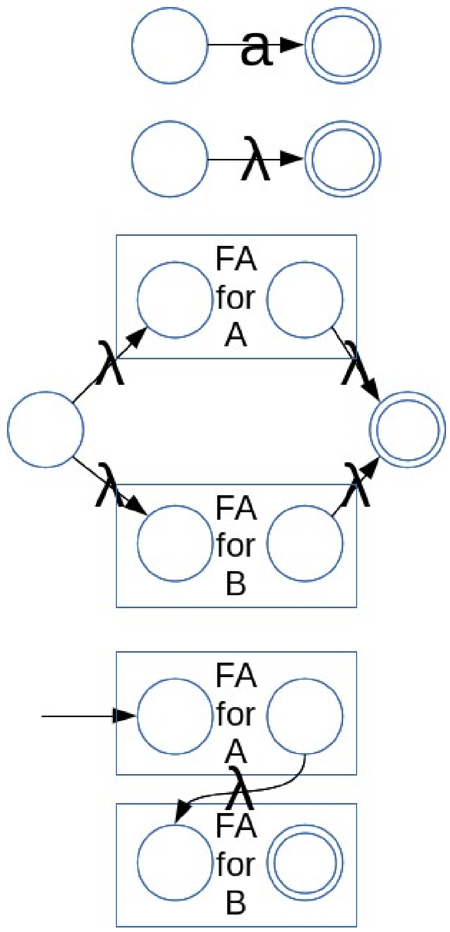
- You can re-write NFA states with two transitions on the same character so that does not happen by introducing a λ-transitions to a new state





Regular Expression -> NFA (2)

- The Rules:
  - Automata for '`a`' and λ are simple
  - Automaton for `(A|B)` has λ-transitions to and from automata for `A` and `B`
  - Automaton for AB has λ-transition between automata `A` and `B`



NFA -> DFA (1)

- Step (2) Apply algorithm:
- The Big Idea:
  - Each DFA state corresponds to the set of NFA states that we could possibly be in
  - We are building D.states and D.transitions
  - We'll use WorkList as a list of states yet to consider
- Algorithm has 3 functions:
  - makeDFA(NFA n)
    - Makes DFA from NFA
  - recordState (NFAStateSet s)
    - Records new DFA state
  - expandToCoverLamdbas(NFAStateSet s)
    - Helper function to recordState() that makes DFA state cover all possible DFA states

NFA -> DFA (2)

`DFA makeDFA(NFA n)`

- `d := new DFA`
- `n.start := recordState({n.start})`
- `foreach ds in WorkList do`
    - `WorkList := WorkList – {ds}`
    - `foreach c in ∑ do`
        - `u := union of all the NFA states reachable by c-transition from an NFA state in DFA state ds`
        - `d.trans(s,c) := recordState(u)`

NFA -> DFA (3)

`DFAState recordState(NFAStateSet ns)`

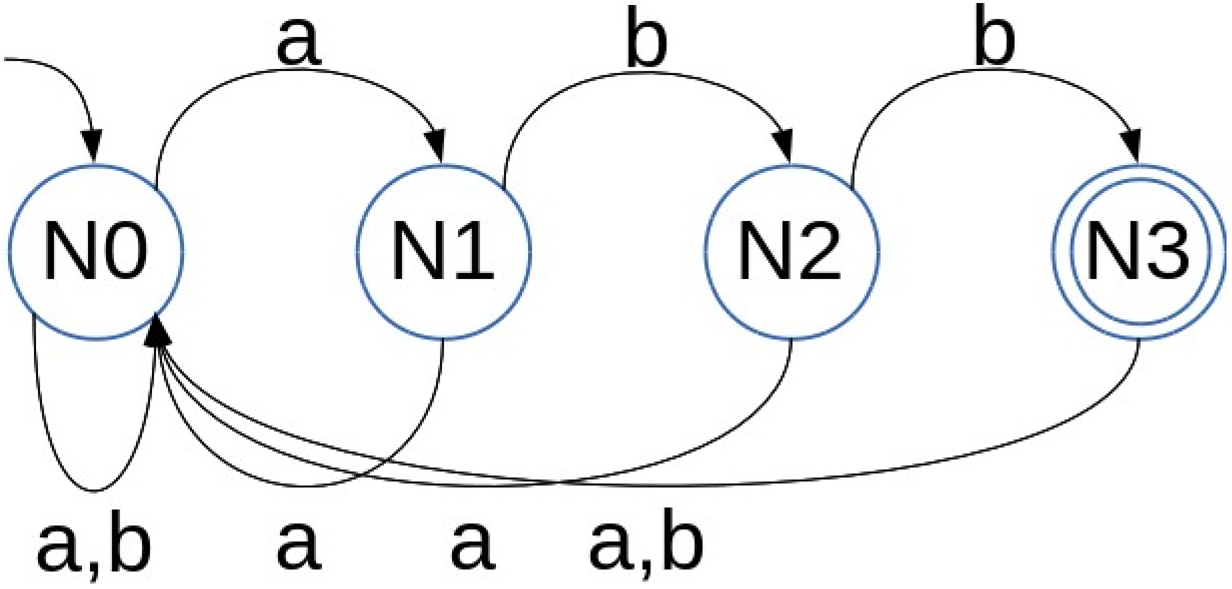
- `ds := expandToCoverLambdas(s)`
- `if ds not in D.states`
  - `D.states := D.states + {ds}`
  - `WorkList := WorkList + {ds}`
- `return ds`

NFA -> DFA (4)

```
DFAState expandToCoverLamdbas(NFAStateSet s)
   – ans := S
   – do
      • changed := false
      • foreach NFA state ns in ans do
         – foreach λ-transition from ns
            • ns2 := NFA.trans(ns,λ)
            • if ns2 not in ans
               • ans := ans + {ns2}
               • changed := true
   – while not changed
   – return ans
```

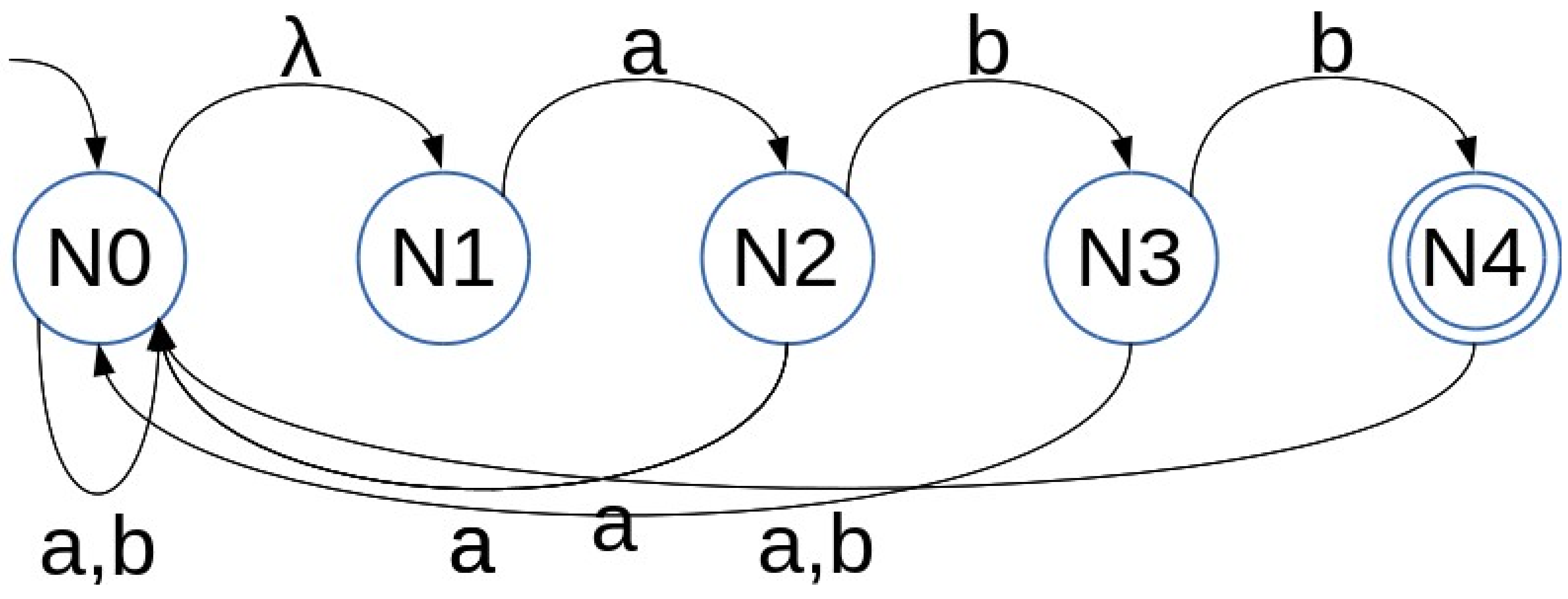
NFA -> DFA, example (1)

- Say you have the following NFA:



NFA -> DFA, example (2)

- Step 1: Invent new nodes with λ transitions to them to get rid of nodes with multiple arcs with same character:



NFA -> DFA, example (3)

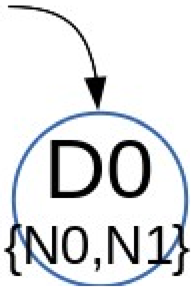
- Initialization:
  - `WorkList := [ ]`
  - `D.states := [ ]`
  - `D.transitions := [ ]`
- Start with start state of NFA: `N0`

NFA -> DFA, example (4)

- **`recordState(s = {N0})`**
  - `s = close({N0})`
  - `s = {N0,N1}` **`// see work to right`**
  - `D.states := [D0 = {N0,N1}]`
  - `WorkList := [{N0,N1}]`
  - `return D0={N0,N1}`
- **`close( {N0} )`**
  - `ans := {N0}`
  - λ-transition `N0->N1` where `N1` not in `ans`
    - `ans := ans + {N1}`
  - No other λ-transitions from `N0` or `N1`
  - So `close({N0}) = {N0,N1}`

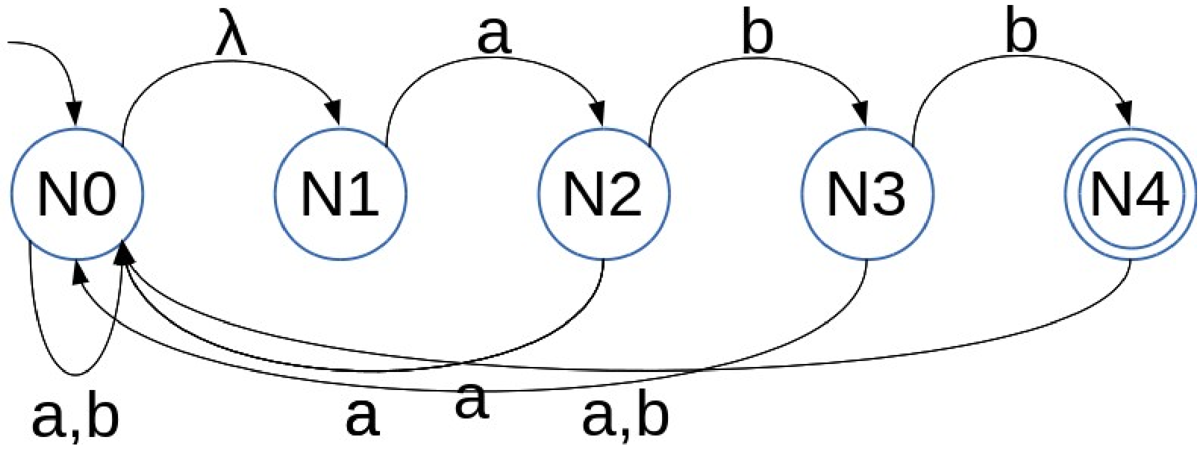
NFA -> DFA, example (5)

- So far



NFA -> DFA, example (6)

- ds := D0(={N0,N1})
- WorkList := [D0] - D0
  - `u :=` union of all the NFA states reachable by 'a'-transition from an NFA state in DFA state `D0={N0,N1}`
  - `u := {N0,N2}`
  - `d.trans(D0,'a') := recordState({N0,N2})`



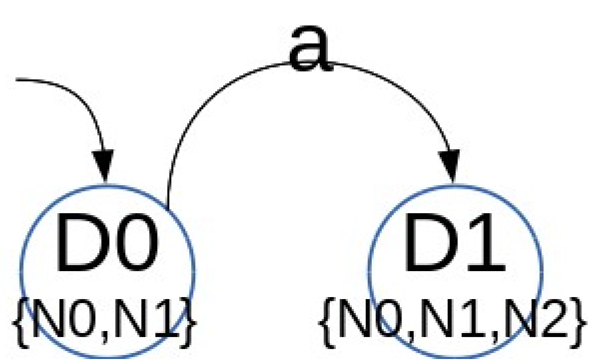
NFA -> DFA, example (7)

- **`recordState(s = {N0,N2})`**
  - `s = close({N0,N2})`
  - `s = {N0,N1,N2}` **`// see work to right`**
  - `D.states := [D1 = {N0,N1,N2}]`
  - `WorkList := [{N0,N1,N2}]`
  - `return D1 = {N0,N1,N2}`

- **`close({N0,N2})`**
  - `ans := {N0,N2}`
  - λ-transition `N0->N1` where `N1` not in `ans`
    - `ans := ans + {N1}`
  - No other λ-transitions from `N0`, `N1` or `N2`
  - So `close({N0,N2})` = `{N0,N1,N2}`

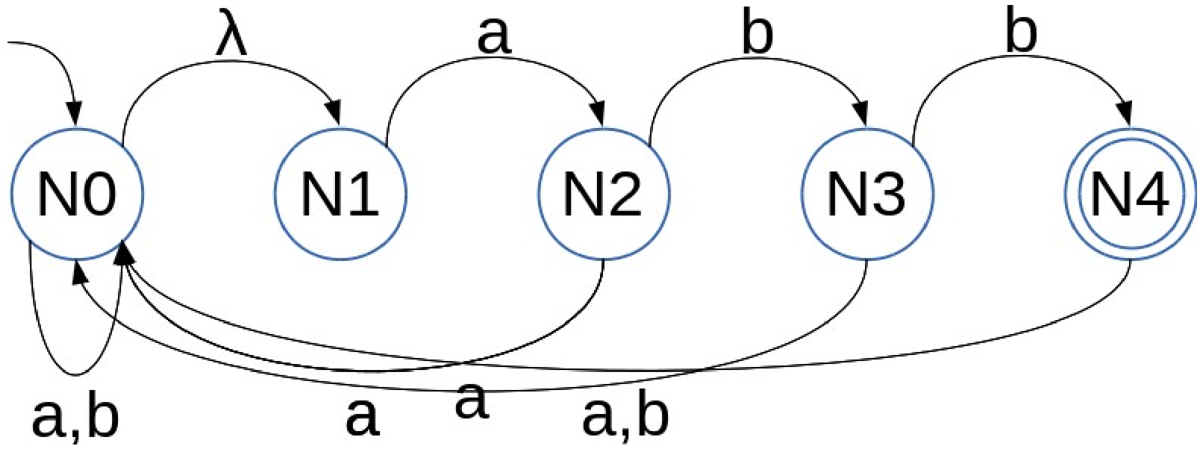
NFA -> DFA, example (8)

- So far



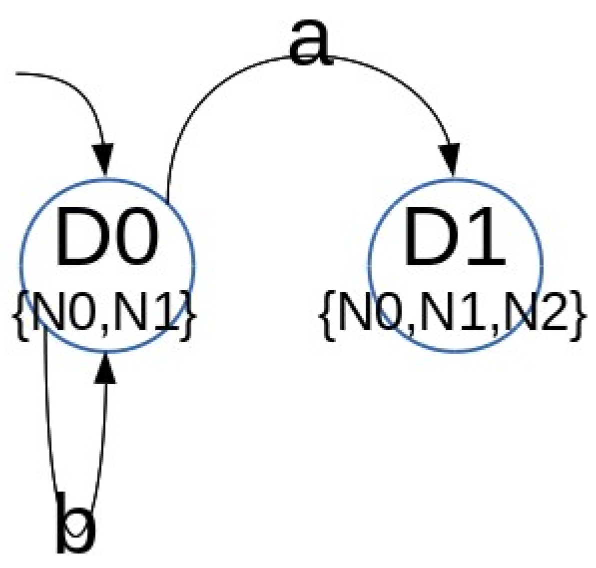
NFA -> DFA, example (9)

- `u :=` union of all the NFA states reachable by 'b'-transition from an NFA state in DFA state `D0={N0,N1}`
- `u := {N0}`
- `d.trans(D0,'b') := recordState({N0}) = D0(={N0,N1})`



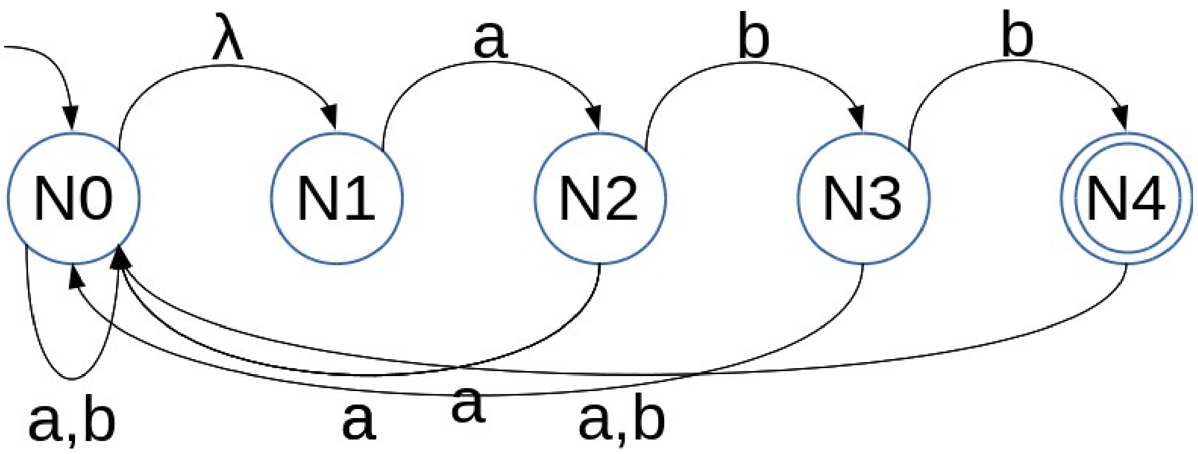
NFA -> DFA, example (10)

- So far



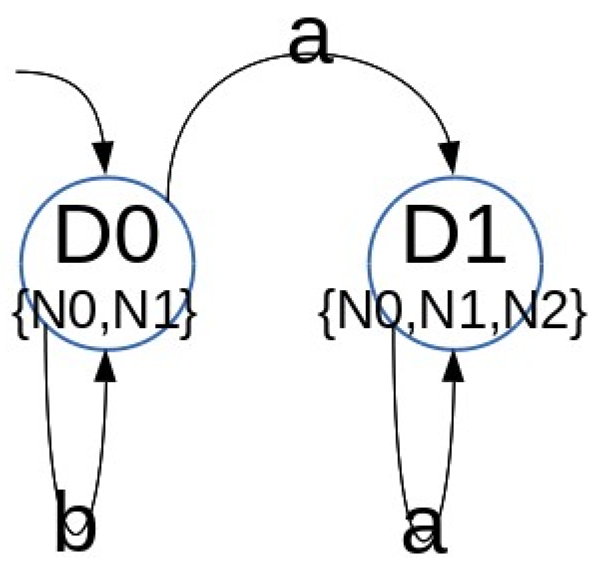
NFA -> DFA, example (11)

- ds := D1(={N0,N1,N2})
- WorkList := [D1] - D1
  - `u` := union of all the NFA states reachable by 'a'-transition from an NFA state in DFA state `D1={N0,N1,N2}`
  - `u := {N0,N2}`
  - `d.trans(D1,'a') := recordState({N0,N2}) = D1(={N0,N1,N2})`



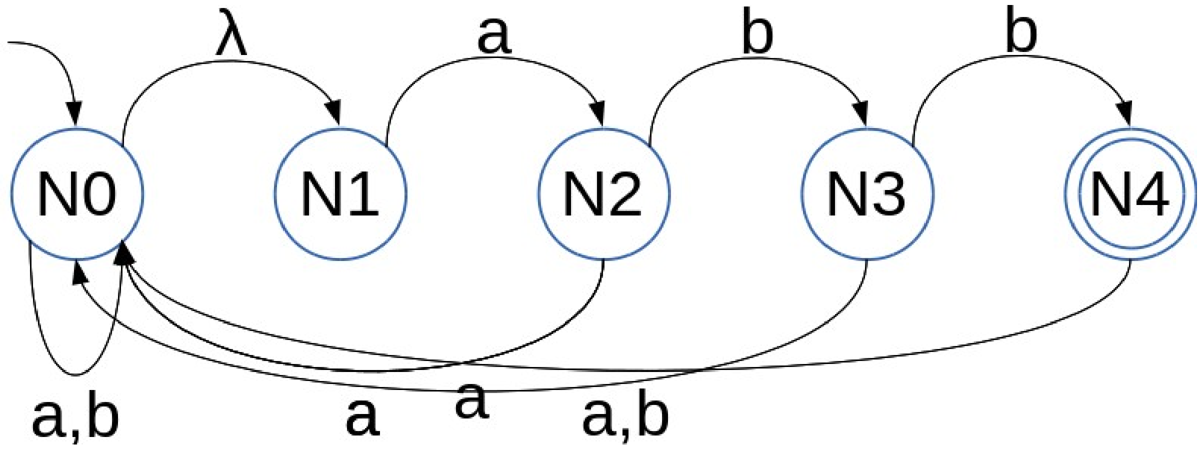
NFA -> DFA, example (12)

- So far



NFA -> DFA, example (13)

- `u :=` union of all the NFA states reachable by 'b'-transition from an NFA state in DFA state `D1={N0,N1,N2}`
- `u := {N0,N3}`
- `d.trans(D1,'b') := recordState({N0,N3})`



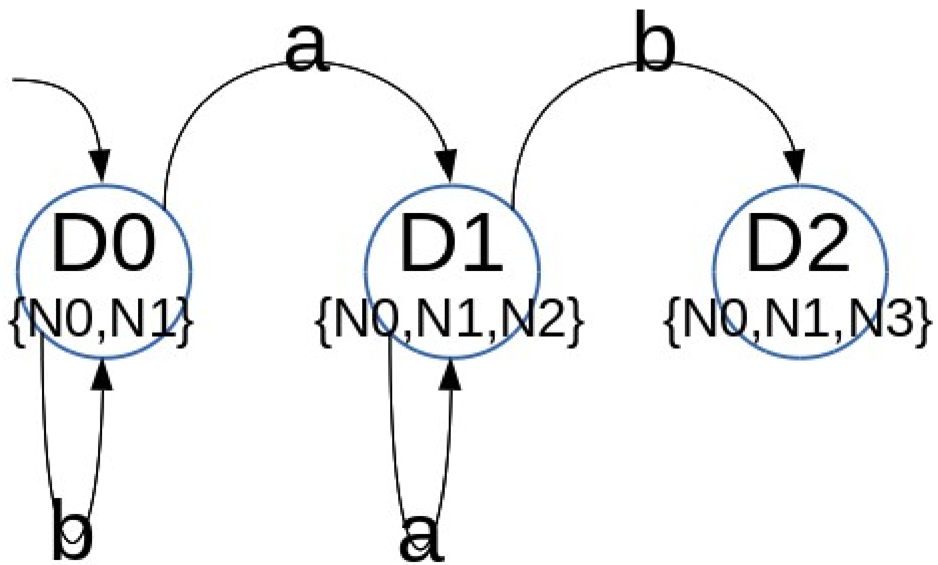
NFA -> DFA, example (14)

- **`recordState(s = {N0,N3})`**
  - `s = close({N0,N3})`
  - `s = {N0,N1,N3}` **`// see work to right`**
  - `D.states := [D2 = {N0,N1,N3}]`
  - `WorkList := [{N0,N1,N3}]`
  - `return D2 = {N0,N1,N3}`

- **`close({N0,N3})`**
  - `ans := {N0,N3}`
  - λ-transition `N0->N1` where `N1` not in `ans`
    - `ans := ans + {N1}`
  - No other λ-transitions from `N0`, `N1` or `N3`
  - So `close({N0,N3})` = `{N0,N1,N3}`

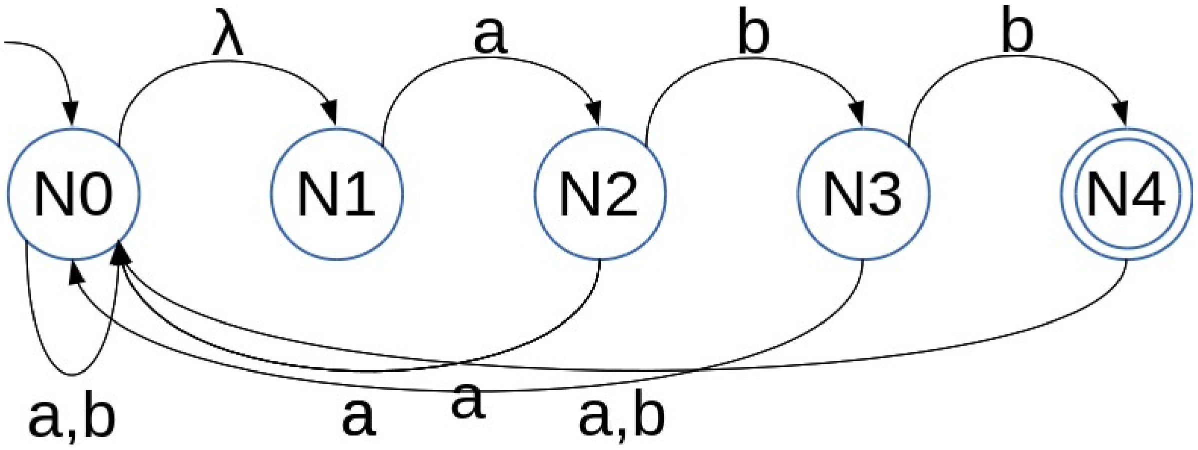
NFA -> DFA, example (15)

- So far



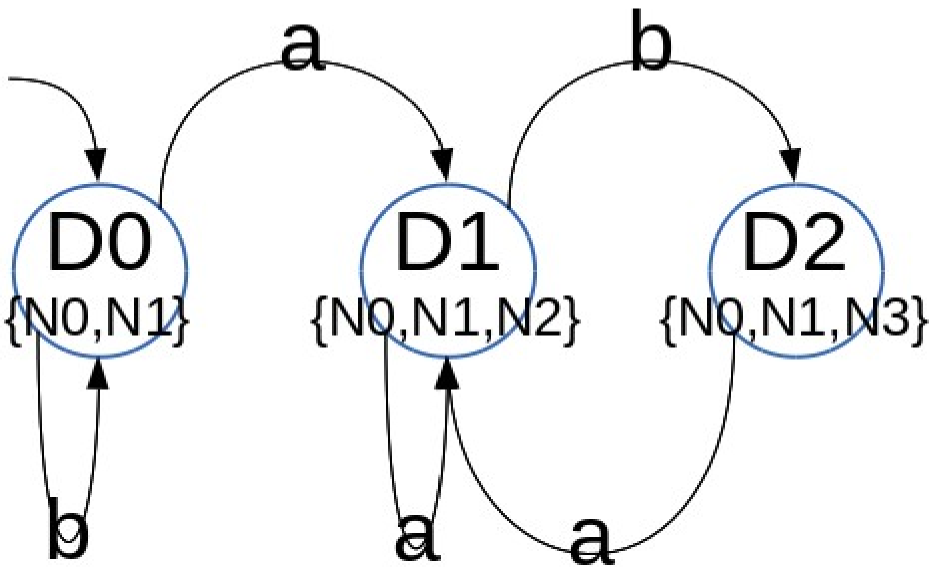
NFA -> DFA, example (16)

- ds := D2(={N0,N1,N3})
- WorkList := [D2] - D2
  - `u :=` union of all the NFA states reachable by 'a'-transition from an NFA state in DFA state `D2={N0,N1,N3}`
  - `u := {N0,N2}`
  - `d.trans(D2,'a') := recordState({N0,N2}) = D1(={N0,N1,N2})`



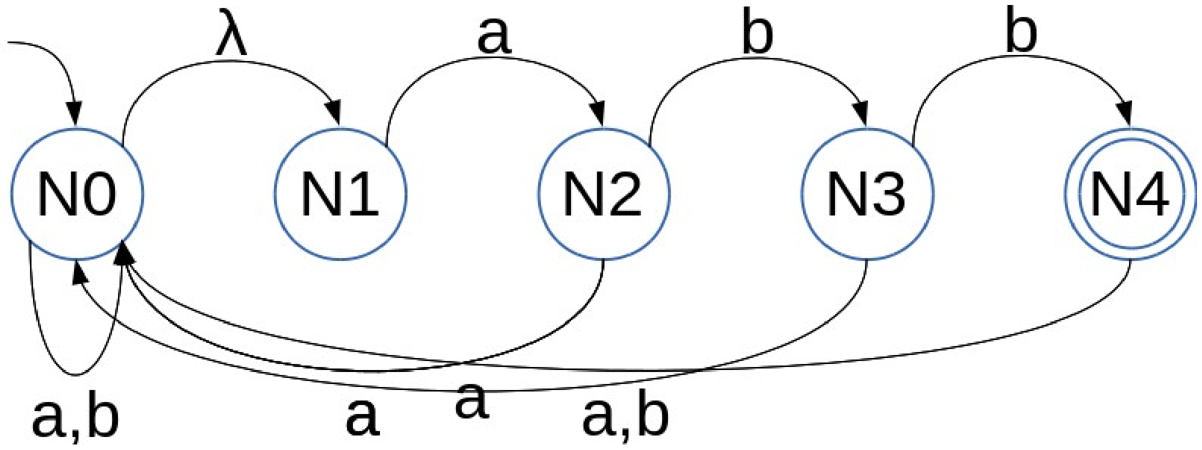
NFA -> DFA, example (17)

- So far



NFA -> DFA, example (18)

- `u :=` union of all the NFA states reachable by 'b'-transition from an NFA state in DFA state `D2={N0,N1,N3}`
- `u := {N0,N4}`
- `d.trans(D2,'b') := recordState({N0,N4})`

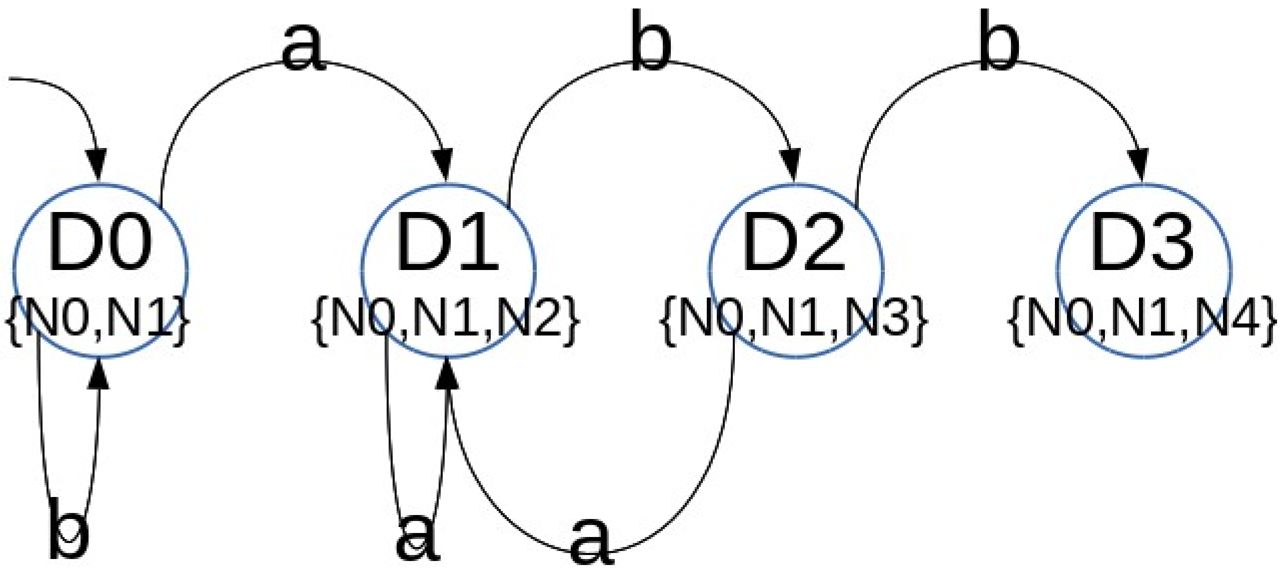


NFA -> DFA, example (19)

- **`recordState(s = {N0,N4})`**
  - `s = close({N0,N4})`
  - `s = {N0,N1,N4}` **`// see work to right`**
  - `D.states := [D3 = {N0,N1,N4}]`
  - `WorkList := [{N0,N1,N4}]`
  - `return D3 = {N0,N1,N4}`
- **`close({N0,N4})`**
  - `ans := {N0,N4}`
  - λ-transition `N0->N1` where `N1` not in `ans`
    - `ans := ans + {N1}`
  - No other λ-transitions from `N0`, `N1` or `N4`
  - So `close({N0,N4}) = {N0,N1,N4}`

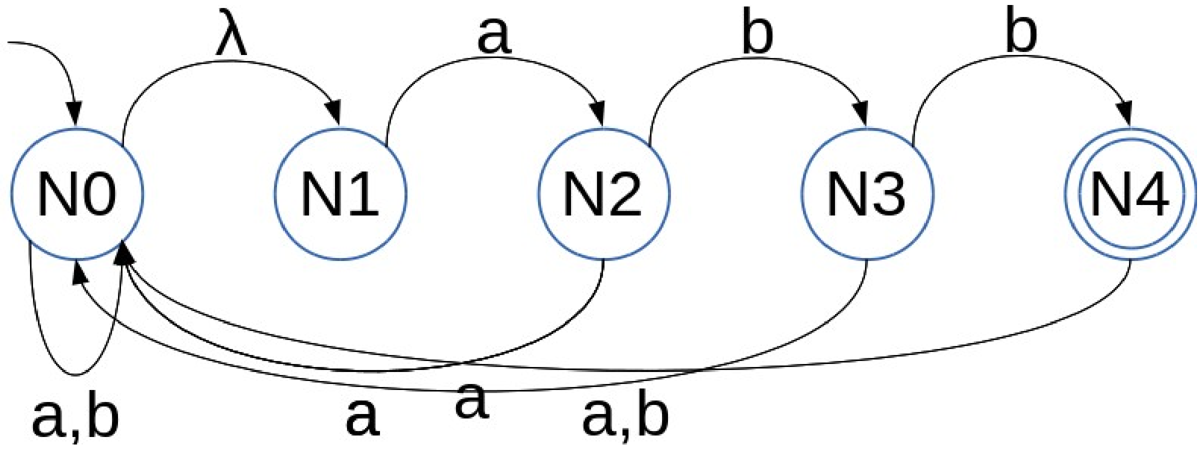
NFA -> DFA, example (20)

- So far



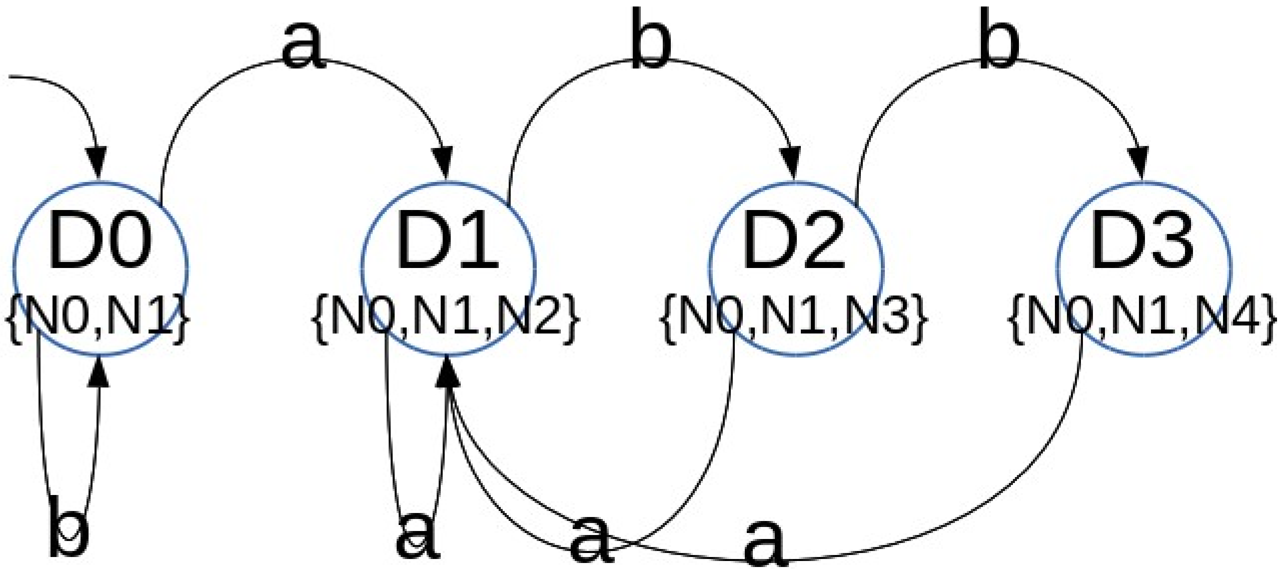
NFA -> DFA, example (21)

- ds := D3(={N0,N1,N4})
- WorkList := [D3] - D3
  - `u :=` union of all the NFA states reachable by 'a'-transition from an NFA state in DFA state `D3={N0,N1,N4}`
  - `u := {N0,N2}`
  - `d.trans(D3,'a') := recordState({N0,N2}) = D1(={N0,N1,N2})`



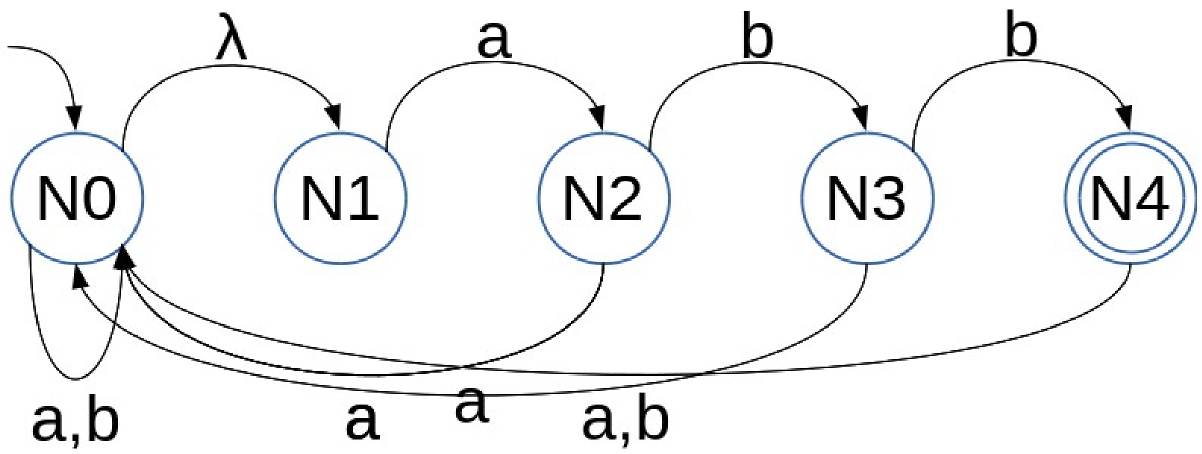
NFA -> DFA, example (22)

- So far



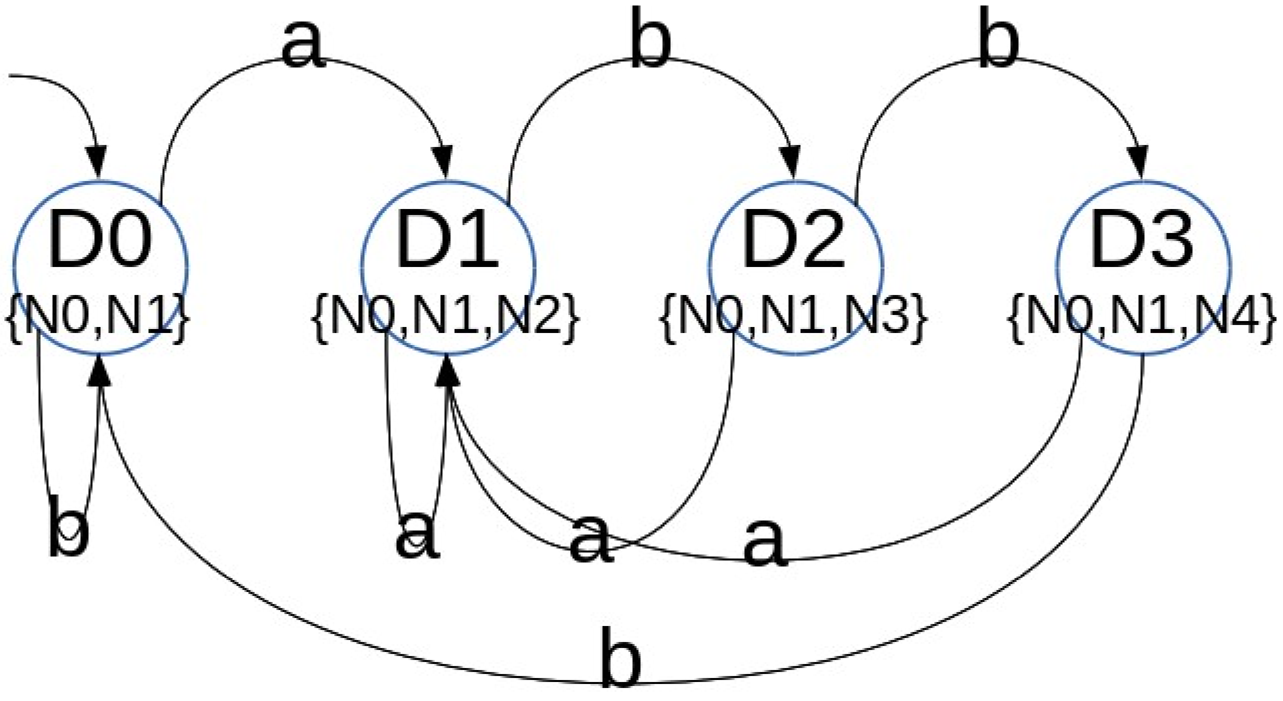
NFA -> DFA, example (23)

- `u :=` union of all the NFA states reachable by 'b'-transition from an NFA state in DFA state `D3={N0,N1,N4}`
- `u := {N0}`
- `d.trans(D3,'b') := recordState({N0}) = D0(={N0,N1})`



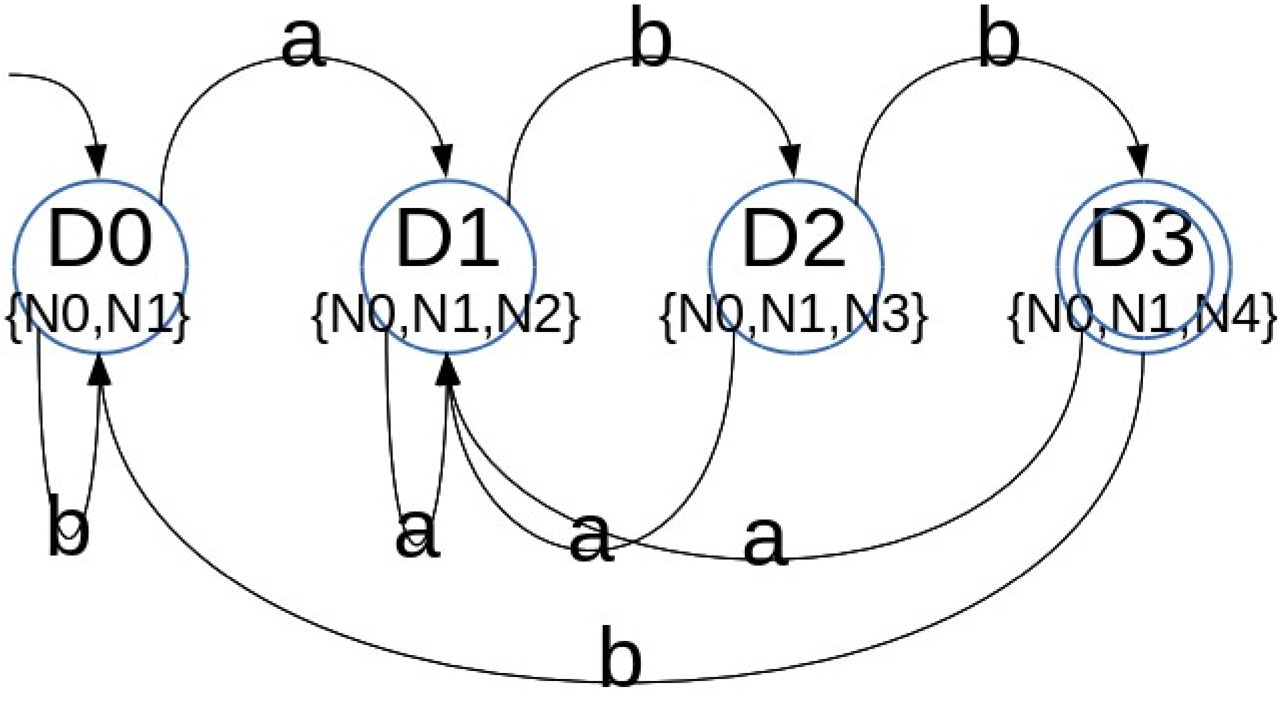
NFA -> DFA, example (24)

- So far



Last Step!

- Any DFA state that has at least one NFA accepting state is itself an accepting state:



Constructing a Scanner

- Coding
  - Table-driven
  - Explicit Control
- Transducers
  - Give the token implied by the string
- Look-ahead
- Reserved words

Practical Issues:

- String literals
- Reserved words
- Pre-processor and compiler directives
- End-of-input
- Look-ahead

References:

- Hopcroft, John; Ullman, Jeffery. "*Introduction to Automata Theory, Languages and Computation*." Addison-Wesley. 1979.