NTNU | Norwegian University of Science and Technology

Compiler Construction

Lecture 4: Lexical analysis in the real world

Michael Engel

Includes material by Jan Christian Meyer

Overview

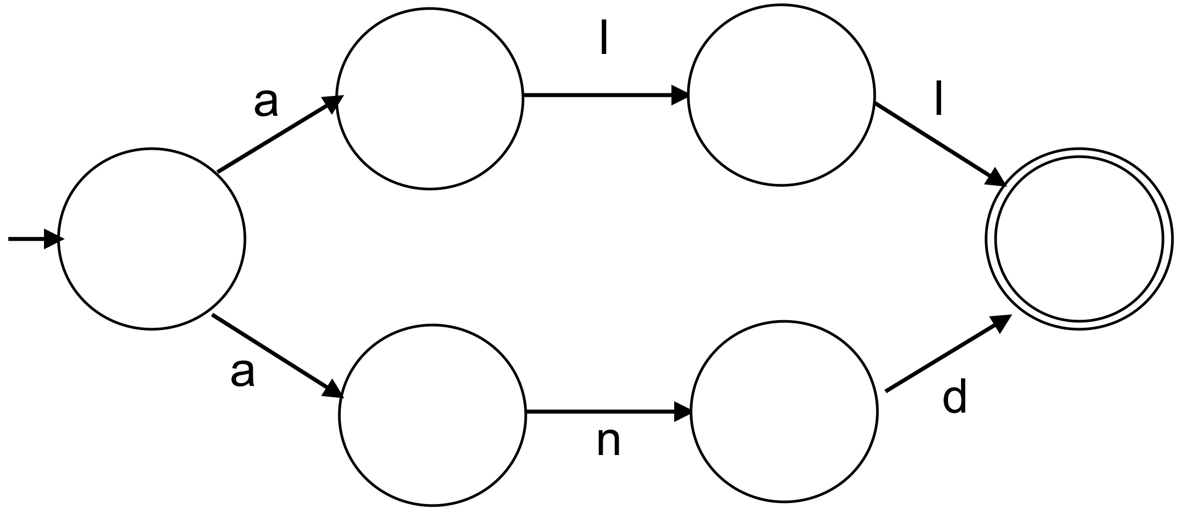
- NFA to DFA conversion
  - Subset construction algorithm
- DFA state minimization:
  - Hopcroft's algorithm
  - Myhill-Nerode method
- Using a scanner generator
  - lex syntax and usage
  - lex examples

What have we achieved so far?

- We know a method to convert a regular expression:

(all | and)

into a ***nondeterministic*** finite automaton (NFA):

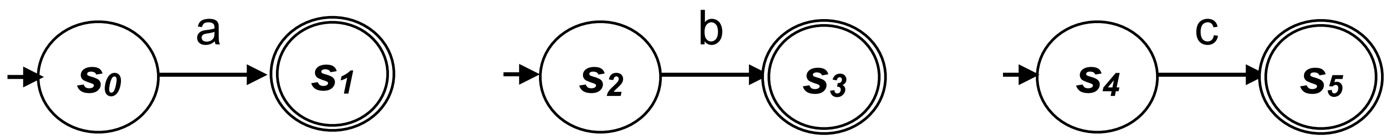


using the McNaughton, Thompson and Yamada algorithm

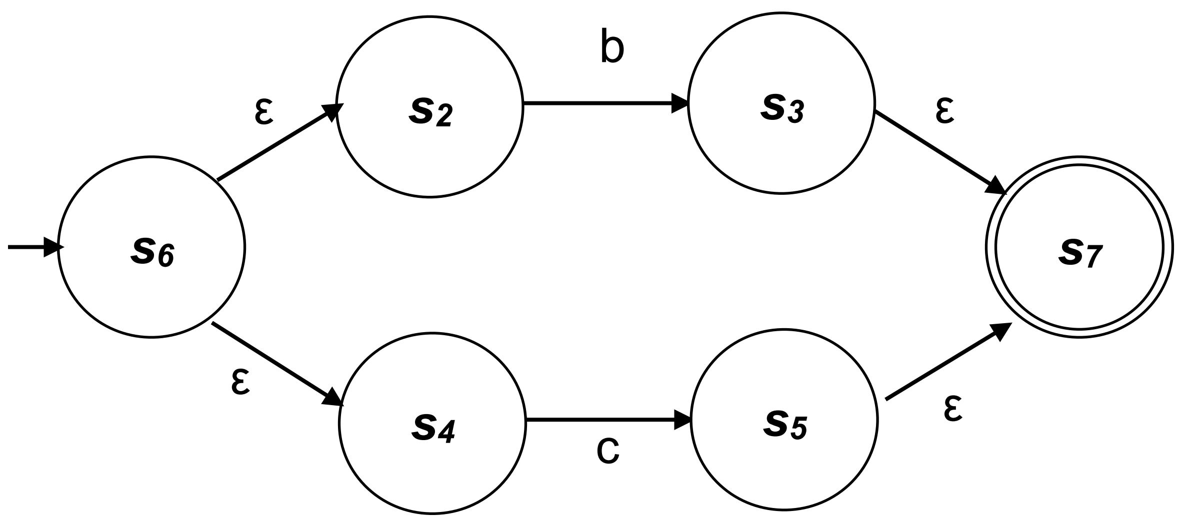
Overhead of constructed NFAs

Let's look at another example: **a(b|c)***

- Construct the simple NFAs for **a**, **b** and **c**

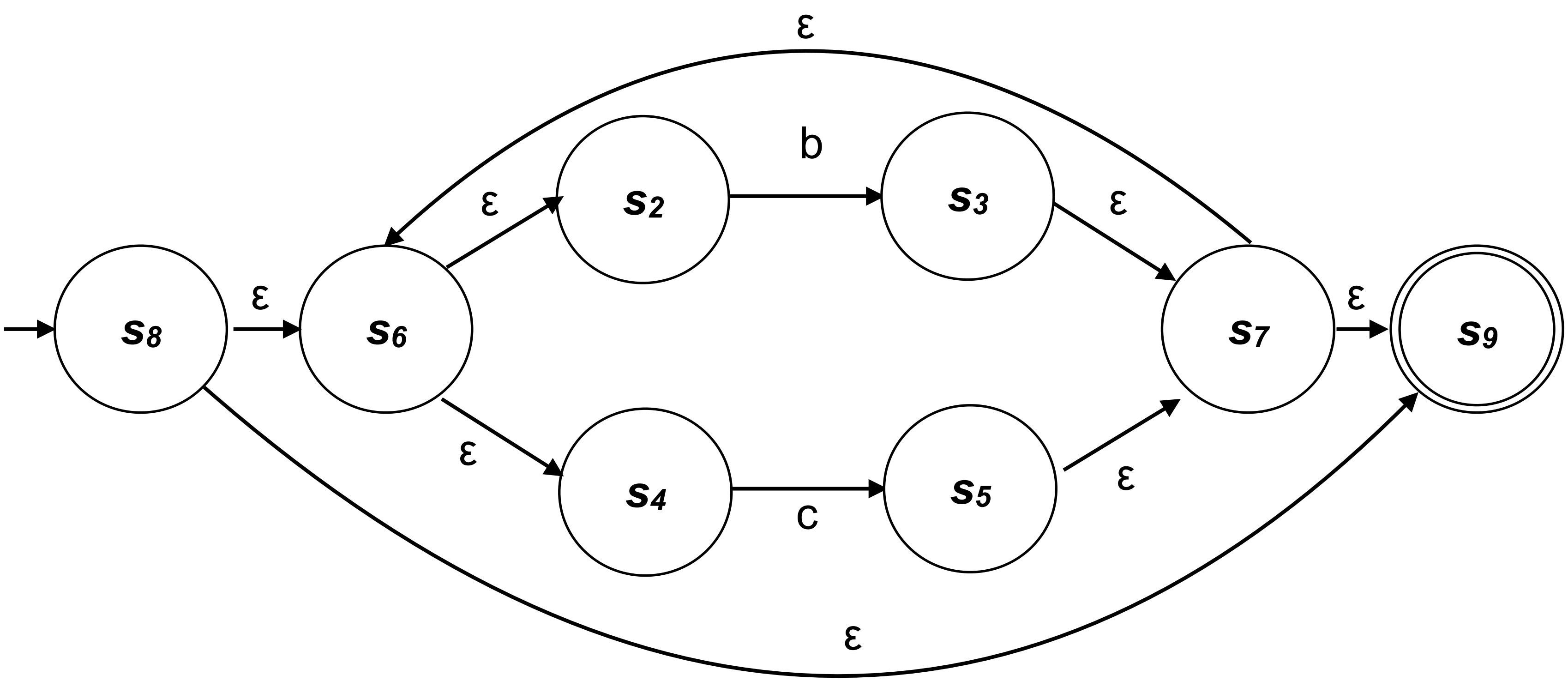


- Construct the NFA for **b|c**



Overhead of constructed NFAs

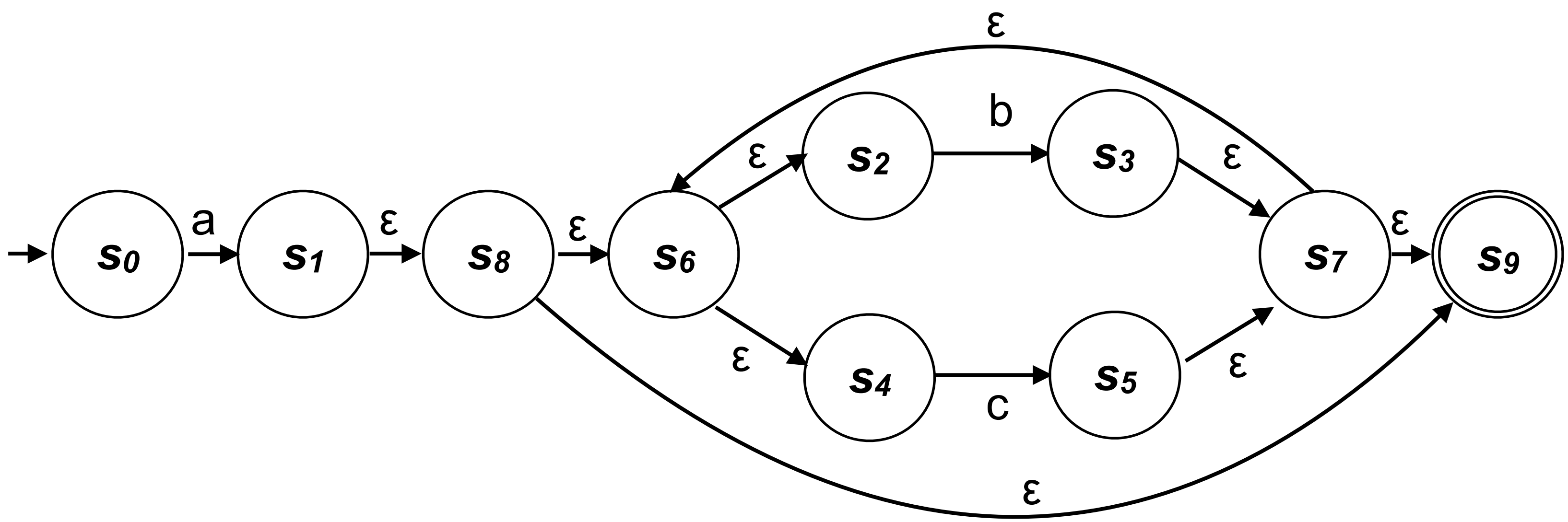
- Now construct the NFA for **(b|c)***



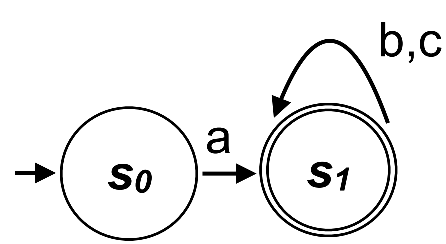
- Looks pretty complex already? We're not even finished…

Overhead of constructed NFAs

- Finally, construct the NFA for **a(b|c)***



- This NFA has many more states than a minimal human-built DFA:



From NFA to DFA

- An NFA is not really helpful
  …since its implementation is not obvious
- We know: every DFA is also an NFA (without ε-transitions)
  - Every NFA can also be converted to an equivalent DFA
    (this can be proven by induction, we just show the construction)
- The method to do this is called ***subset construction:***

NFA: (Q_N, Σ, δ_N, n_0, F_N)

↓

DFA: (Q_D, Σ, δ_D, d_0, F_D)

The alphabet Σ stays the same

The set of states Q_N,
transition function δ_N,
start state q_{N0}
and set of accepting states F_N
are modified

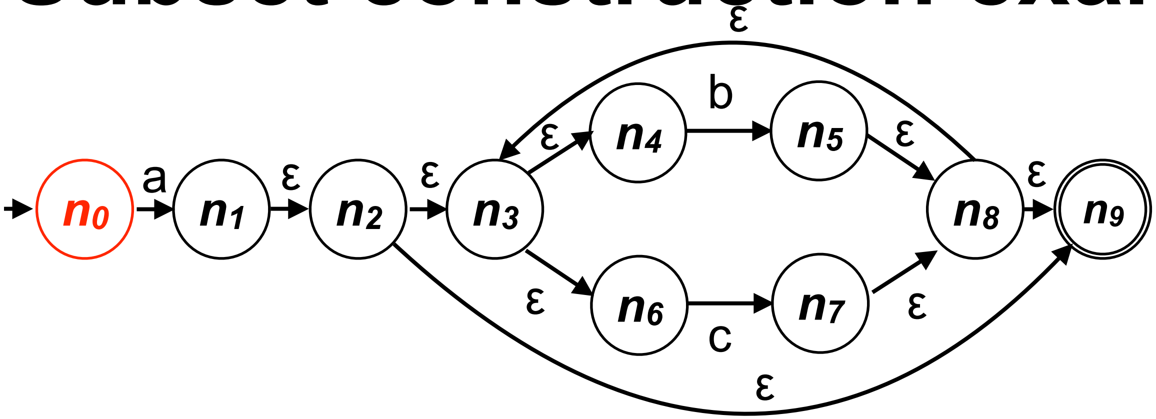
Subset construction algorithm

```
q₀ ← ε-closure({n₀});
Q_D ← q₀;
WorkList ← {q₀};

while (WorkList != ∅) do
  remove q from WorkList;
  for each character c∈Σ do
    t ← ε-closure(δ_N(q,c));
    δ_D[q,c] ← t;
    if t ∉ Q_D then
      add t to Q_D and to WorkList;
  end;
end;
```

Idea of the algorithm:
Find sets of states that are equivalent (due to ε-transitions) and join these to form states of a DFA

ε-closure:
contains a set of states ***S*** and any states in the NFA that can be reached from one of the states in ***S*** along paths that contain only ε-transitions (these are identical to a state in ***S***)

Subset construction example

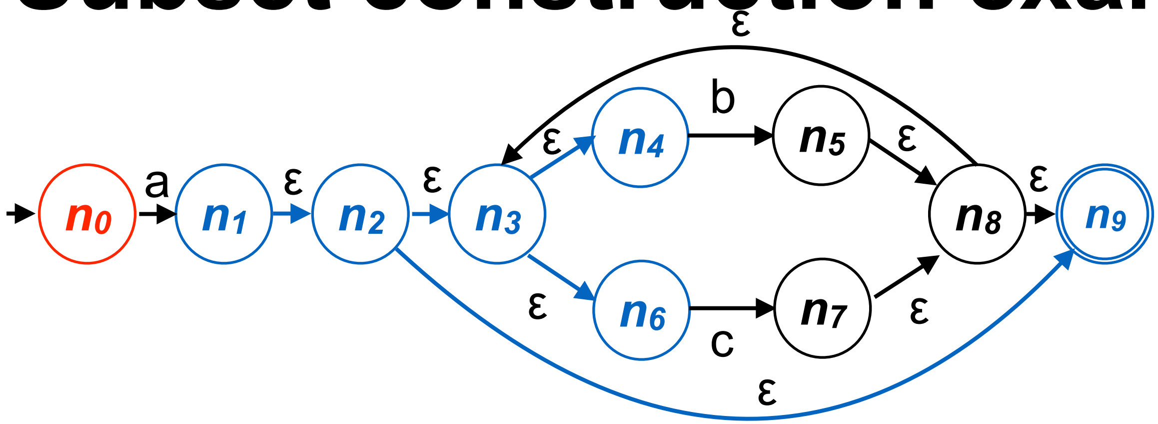


δ_N	a	b	c	ε
n_0	n_1	–	–	–
n_1	–	–	–	n_2
n_2	–	–	–	n_3, n_9
n_3	–	–	–	n_4, n_6
n_4	–	n_5	–	–
n_5	–	–	–	n_8
n_6	–	–	n_7	–
n_7	–	–	–	n_8
n_8	–	–	–	n_3, n_9
n_9	–	–	–	–

```
q0 ← {n0}
QD ← {n0};
WorkList ← {n0};
```

```
q0 ← ε-closure({n0});
QD ← q0;
WorkList ← {q0};

while (WorkList != ∅) do
  remove q from WorkList;
  for each character c∈Σ do
    t ← ε-closure(δN(q,c));
    δD[q,c] ← t;
    if t ∉ QD then
      add t to QD and to WorkList;
  end;
end;
```

Subset construction example

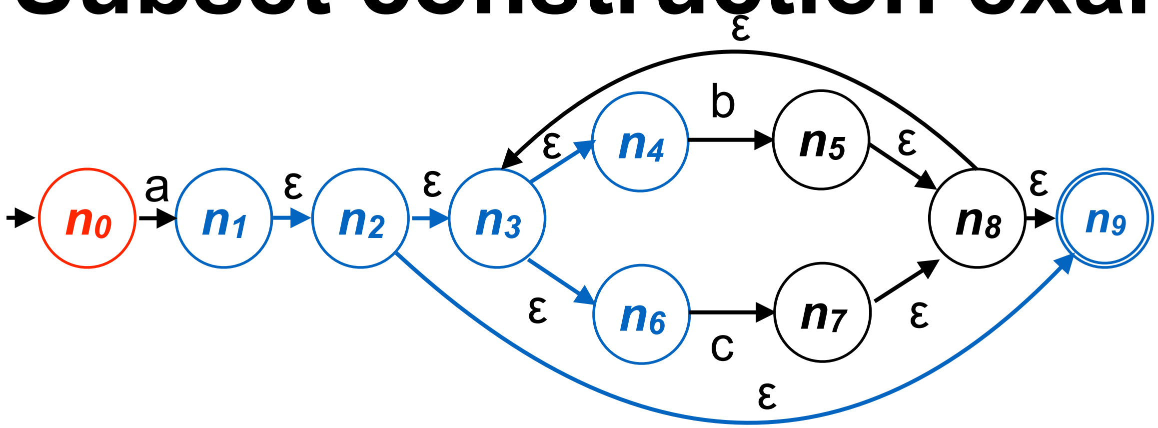



```
q0 ← ε-closure({n0});
QD ← q0;
WorkList ← {q0};

while (WorkList != ∅) do
  remove q from WorkList;
  for each character c∈Σ do
    t ← ε-closure(δN(q,c));
    δD[q,c] ← t;
    if t ∉ QD then
      add t to QD and to WorkList;
  end;
end;
```

δ_N	a	b	c	ε
n_0	n_1	–	–	–
n_1	–	–	–	n_2
n_2	–	–	–	n_3,n_9
n_3	–	–	–	n_4,n_6
n_4	–	n_5	–	–
n_5	–	–	–	n_8
n_6	–	–	n_7	–
n_7	–	–	–	n_8
n_8	–	–	–	n_3,n_9
n_9	–	–	–	–

```
while-loop Iteration 1
WorkList ← {{n0}};
q ← n0;
c ← 'a':
   t ← ε-closure(δN(q,c))
     = ε-closure(δN(n0,'a'))
     = ε-closure(n1)
     = {n1,n2,n3,n4,n6,n9}
   δD[n0,'a']←{n1,n2,n3,n4,n6,n9};
   QD ←{{n0},{n1,n2,n3,n4,n6,n9}};
   WorkList ←
            {{n1,n2,n3,n4,n6,n9}};
```

Subset construction example



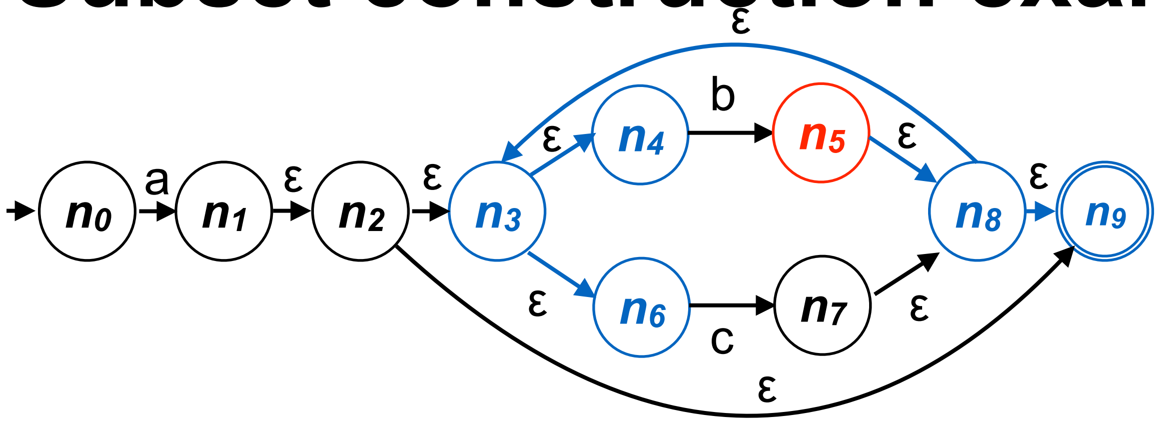
δ_N	a	b	c	ε
n_0	n_1	–	–	–
n_1	–	–	–	n_2
n_2	–	–	–	n_3,n_9
n_3	–	–	–	n_4,n_6
n_4	–	n_5	–	–
n_5	–	–	–	n_8
n_6	–	–	n_7	–
n_7	–	–	–	n_8
n_8	–	–	–	n_3,n_9
n_9	–	–	–	–

```
while-loop Iteration 1:

WorkList ← {n0};
q ← n0;
c ← 'b','c':
   t ← {}
   no change to QD, Worklist
```

```
q0 ← ε-closure({n0});
QD ← q0;
WorkList ← {q0};

while (WorkList != ∅) do
  remove q from WorkList;
  for each character c∈Σ do
    t ← ε-closure(δN(q,c));
    δD[q,c] ← t;
    if t ∉ QD then
      add t to QD and to WorkList;
  end;
end;
```

We will skip the iterations of the for loop that do not change Q_D from now on

Subset construction example

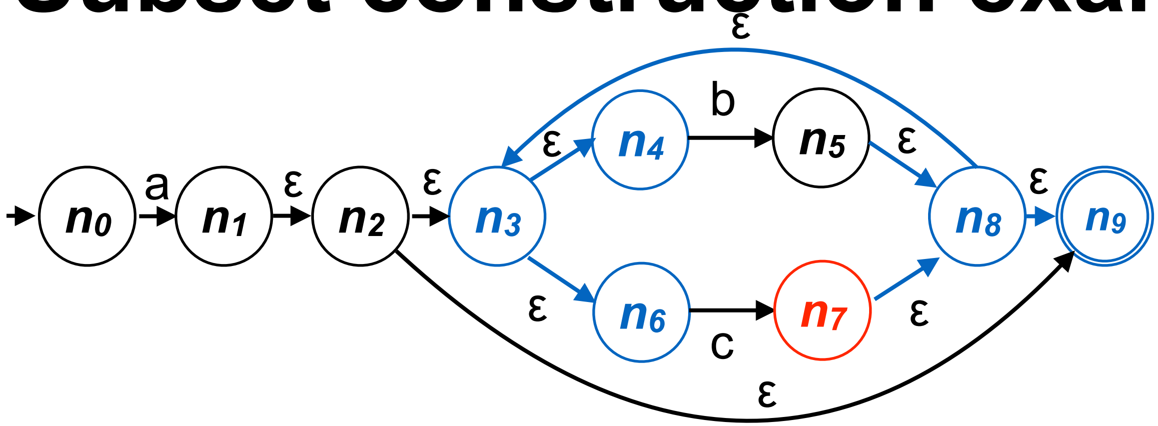


δ_N	a	b	c	ε
n_0	n_1	–	–	–
n_1	–	–	–	n_2
n_2	–	–	–	n_3,n_9
n_3	–	–	–	n_4,n_6
n_4	–	n_5	–	–
n_5	–	–	–	n_8
n_6	–	–	n_7	–
n_7	–	–	–	n_8
n_8	–	–	–	n_3,n_9
n_9	–	–	–	–

```
while-loop Iteration 2
WorkList = {{n1,n2,n3,n4,n6,n9}};
q ← {n1,n2,n3,n4,n6,n9};
c ← 'b':
  t ← ε-closure(δN(q,c))
    = ε-closure(δN(q,'b'))
    = ε-closure(n5)
    = {n5,n8,n9,n3,n4,n6}
  δD[q,'b']←{n5,n8,n9,n3,n4,n6};
  QD ←{{n0},{n1,n2,n3,n4,n6,n9},
              {n5,n8,n9,n3,n4,n6}};
  WorkList ←
             {{n5,n8,n9,n3,n4,n6}};
```

```
q0 ← ε-closure({n0});
QD ← q0;
WorkList ← {q0};

while (WorkList != ∅) do
  remove q from WorkList;
  for each character c∈Σ do
    t ← ε-closure(δN(q,c));
    δD[q,c] ← t;
    if t ∉ QD then
      add t to QD and to WorkList;
  end;
end;
```

Subset construction example



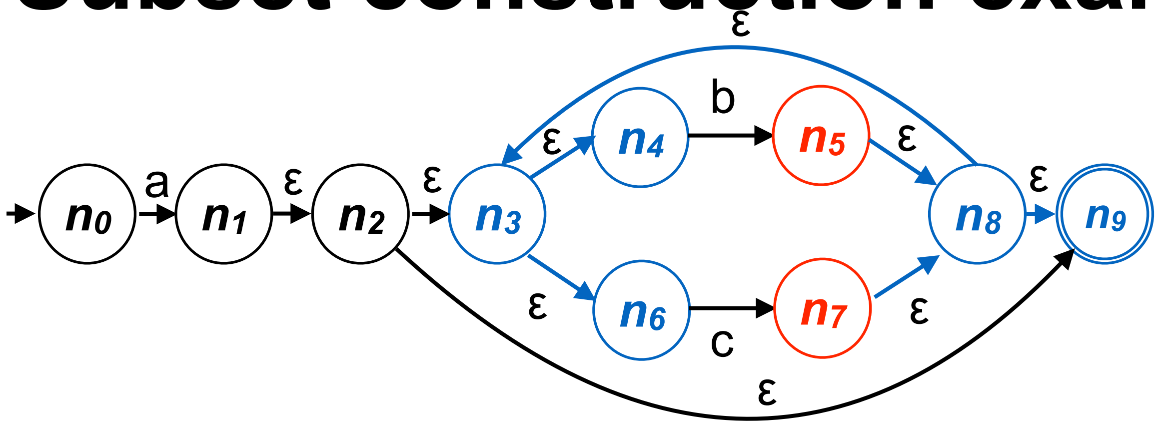

```
q_0 ← ε-closure({n_0});
Q_D ← q_0;
WorkList ← {q_0};

while (WorkList != ∅) do
  remove q from WorkList;
  for each character c∈Σ do
    t ← ε-closure(δ_N(q,c));
    δ_D[q,c] ← t;
    if t ∉ Q_D then
      add t to Q_D and to WorkList;
  end;
end;
```

δ_N	a	b	c	ε
n_0	n_1	–	–	–
n_1	–	–	–	n_2
n_2	–	–	–	n_3,n_9
n_3	–	–	–	n_4,n_6
n_4	–	n_5	–	–
n_5	–	–	–	n_8
n_6	–	–	n_7	–
n_7	–	–	–	n_8
n_8	–	–	–	n_3,n_9
n_9	–	–	–	–

<u>while-loop Iteration 2</u>

```
WorkList = {{n_1,n_2,n_3,n_4,n_6,n_9}};
q ← {n_1,n_2,n_3,n_4,n_6,n_9};
c ← 'c':
   t ← ε-closure(δ_N(q,c))
      = ε-closure(δ_N(q,'c'))
      = ε-closure(n_7)
      = {n_7,n_8,n_9,n_3,n_4,n_6}
   δ_D[q,'c']←{n_7,n_8,n_9,n_3,n_4,n_6};
   Q_D ←{{n_0},{n_1,n_2,n_3,n_4,n_6,n_9},
              {n_5,n_8,n_9,n_3,n_4,n_6},
              {n_7,n_8,n_9,n_3,n_4,n_6}};
   WorkList ←
             {{n_7,n_8,n_9,n_3,n_4,n_6}};
```

Subset construction example



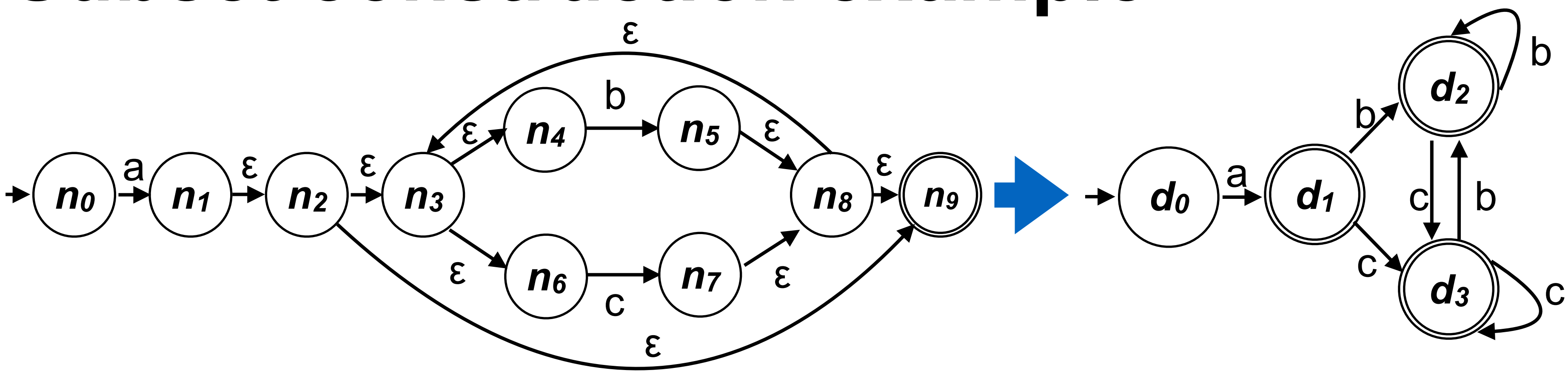

```
q0 ← ε-closure({n0});
QD ← q0;
WorkList ← {q0};

while (WorkList != ∅) do
  remove q from WorkList;
  for each character c∈Σ do
    t ← ε-closure(δN(q,c));
    δD[q,c] ← t;
    if t ∉ QD then
      add t to QD and to WorkList;
  end;
end;
```

δ_N	a	b	c	ε
n_0	n_1	–	–	–
n_1	–	–	–	n_2
n_2	–	–	–	n_3,n_9
n_3	–	–	–	n_4,n_6
n_4	–	n_5	–	–
n_5	–	–	–	n_8
n_6	–	–	n_7	–
n_7	–	–	–	n_8
n_8	–	–	–	n_3,n_9
n_9	–	–	–	–

```
while-loop Iteration 3
WorkList = {{n7,n8,n9,n3,n4,n6}};
q ← {n7,n8,n9,n3,n4,n6};
c ← 'b','c':
  t ← ε-closure(δN(q,c))
    = ε-closure(δN(q,'c'))
    = ε-closure(n5,n7)
// we ran around the graph once!
```

No new states are added to Q_D in this and the following iteration!

Subset construction example

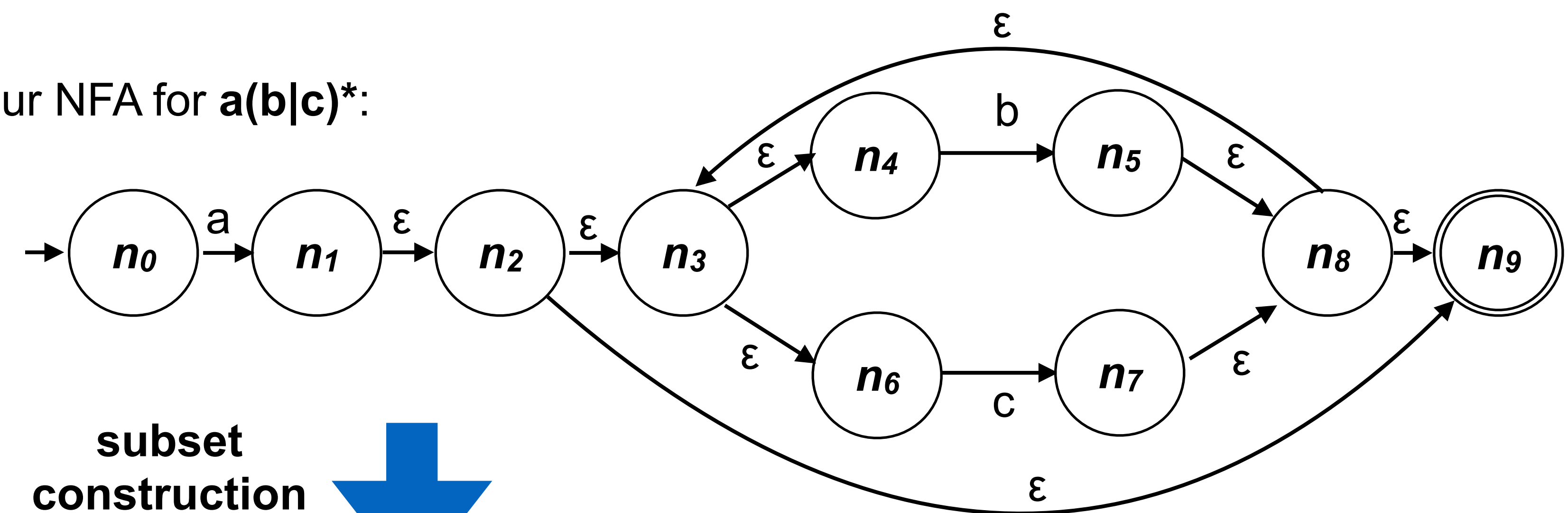


δ_N	a	b	c	ε
n_0	n_1	–	–	–
n_1	–	–	–	n_2
n_2	–	–	–	n_3,n_9
n_3	–	–	–	n_4,n_6
n_4	–	n_5	–	–
n_5	–	–	–	n_8
n_6	–	–	n_7	–
n_7	–	–	–	n_8
n_8	–	–	–	n_3,n_9
n_9	–	–	–	–

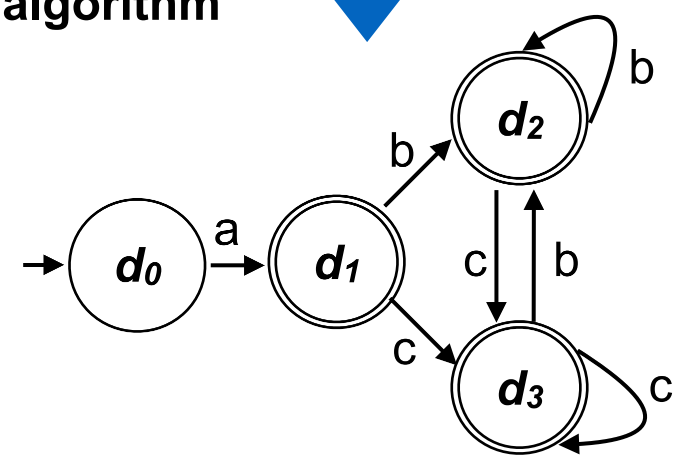
Set name	DFA states	NFA states	ε-closure(δN(q,*))		
			a	b	c
q_0	d_0	n_0	{ $n_1, n_2, n_3, n_4, n_6, n_9$ }	–	–
q_1	d_1	{ $n_1, n_2, n_3, n_4, n_6, n_9$ }	–	{ $n_5, n_8, n_9, n_3, n_4, n_6$ }	{ $n_7, n_8, n_9, n_3, n_4, n_6$ }
q_2	d_2	{ $n_5, n_8, n_9, n_3, n_4, n_6$ }	–	q_2	q_3
q_3	d_3	{ $n_7, n_8, n_9, n_3, n_4, n_6$ }	–	q_2	q_3

Subset construction: result

Our NFA for **a(b|c)***:

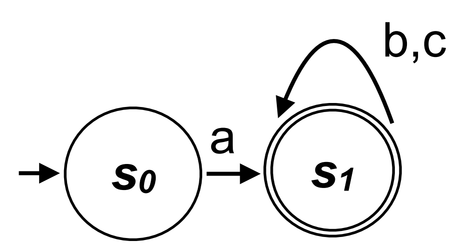


subset construction algorithm



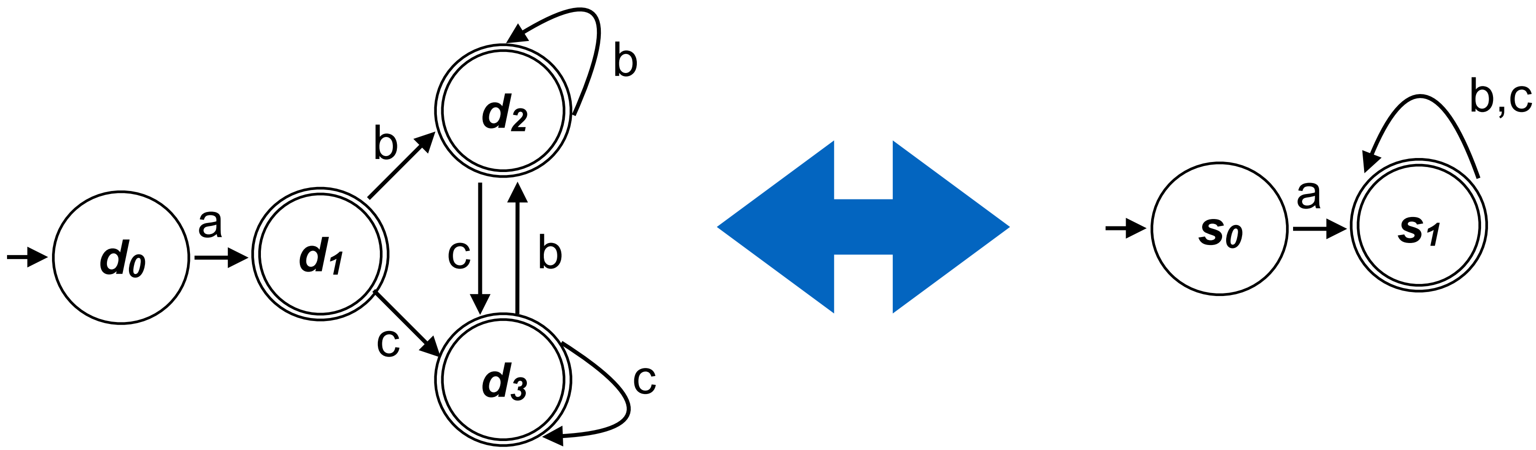
constructed DFA

still bigger than



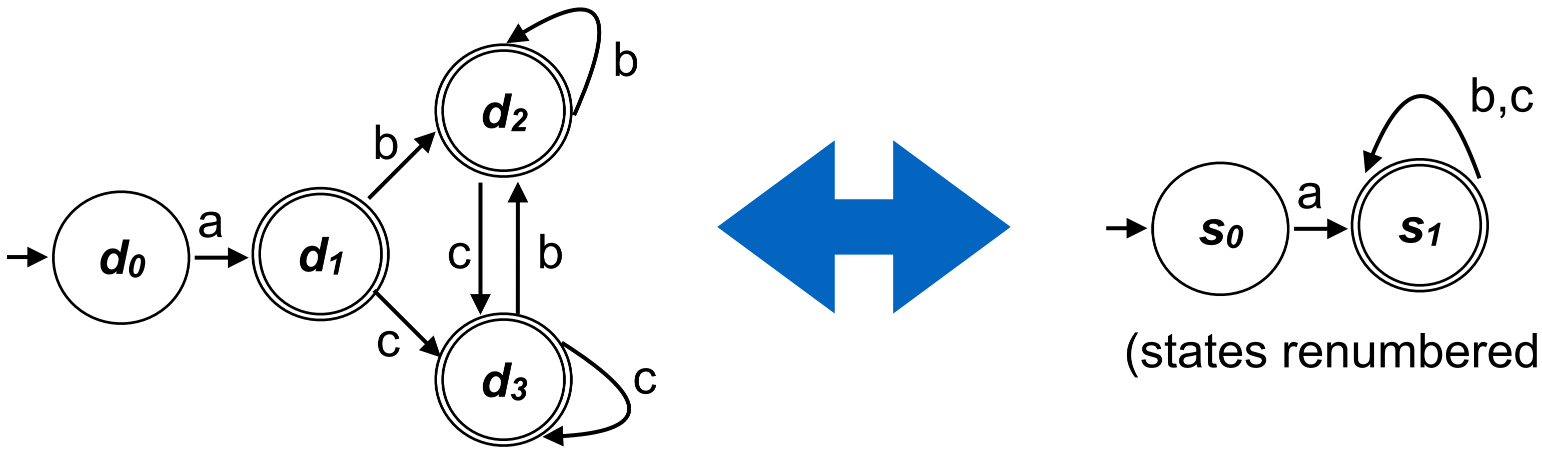
minimal DFA

Minimization of DFAs



- DFAs resulting from subset construction can have a large set of states
  - This ***does not*** increase the time needed to scan a string
  - It ***does*** increase the size of the recognizer in memory
  - On modern computers, the speed of memory accesses often governs the speed of computation
  - A smaller recognizer may fit better into the processor’s cache memory

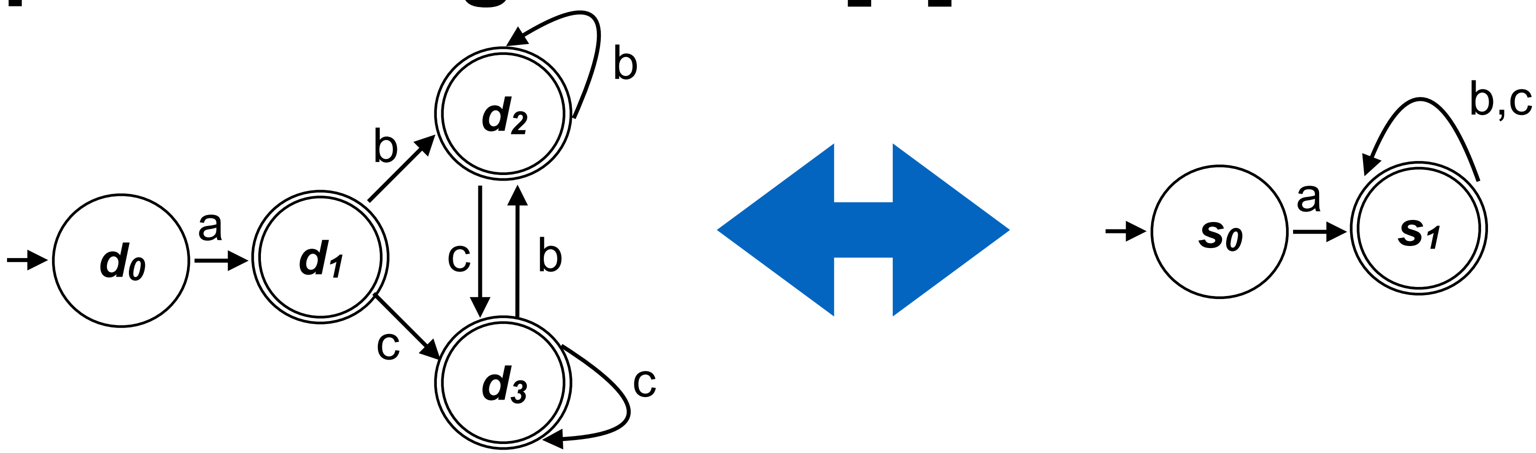
Minimization of DFAs



(states renumbered)

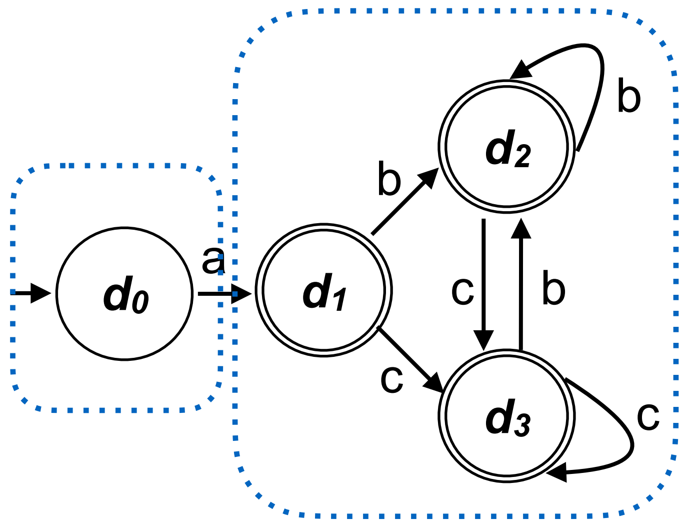
- We need a technique to detect when two states are equivalent
  - i.e. when they produce the same behavior on any input string
- Hopcroft's algorithm [3]
  - finds equivalence classes of DFA states based on their behavior
  - from equivalence classes we can construct a minimal DFA
- We just give an intuitive overview, for details see [4], ch. 2.4.4

Hopcroft’s algorithm [3]



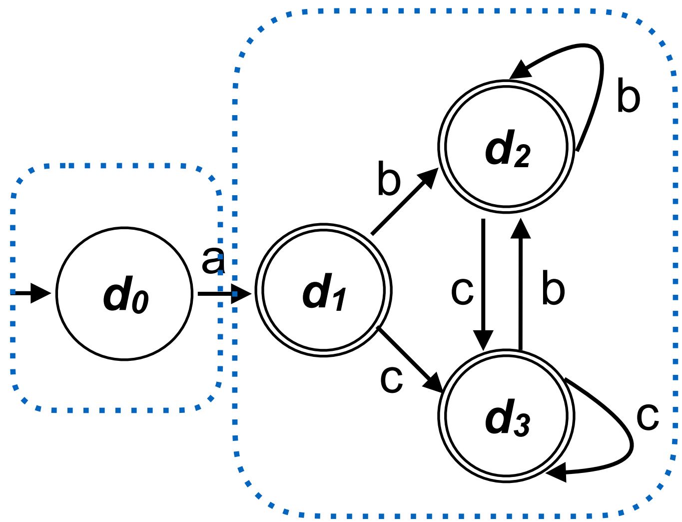
- **Idea:**
  - Two DFA states are equivalent if it's impossible to tell from accepting/rejecting behavior alone which of them the DFA is in
  - For each language, the minimum DFA accepting that language has no equivalent states
- Hopcroft's algorithm works by computing the equivalence classes of the states of the unminimized DFA
- The nub of this computation is an iteration where, at each step, we have a partition of the states that is coarser than equivalence (i.e., equivalent states always belong to the same set of the partition)

Hopcroft's algorithm



1. The initial partition is accepting states and rejecting states. Clearly these are not equivalent

Hopcroft's algorithm

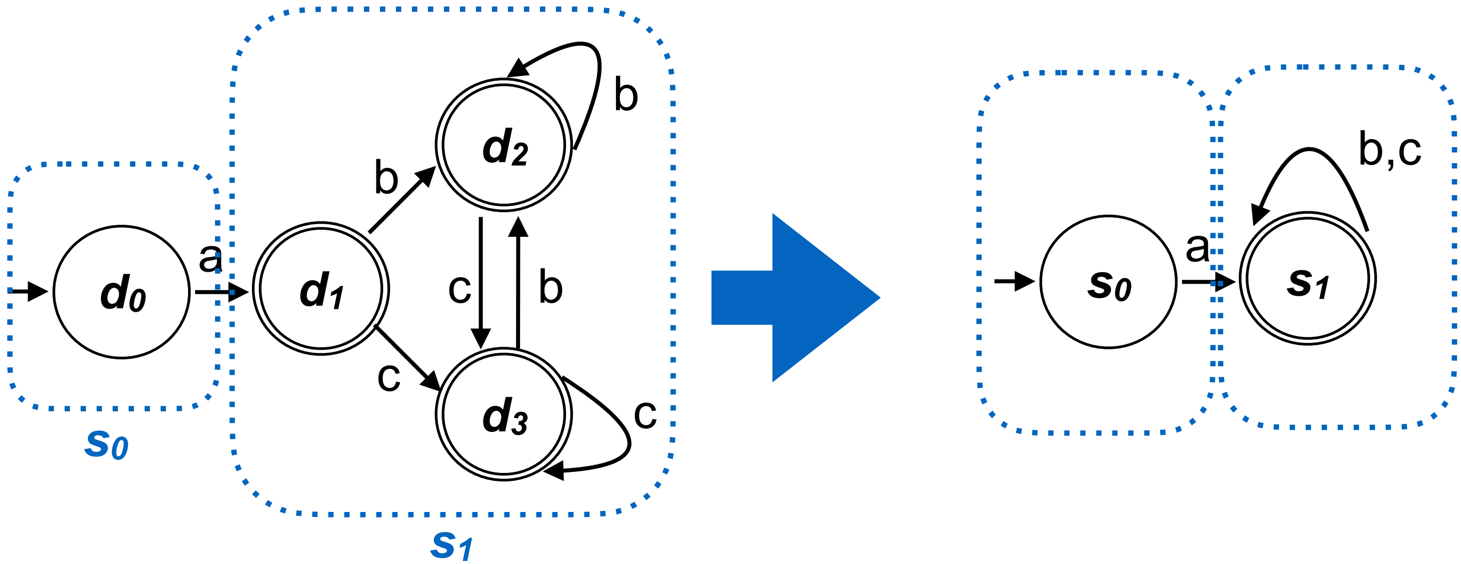


2. Suppose that we have states q1 and q2 in the same set of the current partition:

   If there exists a symbol ***s*** such that δ(q1, ***s***) and δ(q2, ***s***) are in different sets of the partition, then these states are not equivalent

   ⇒ split set of states into subsets of equivalent states

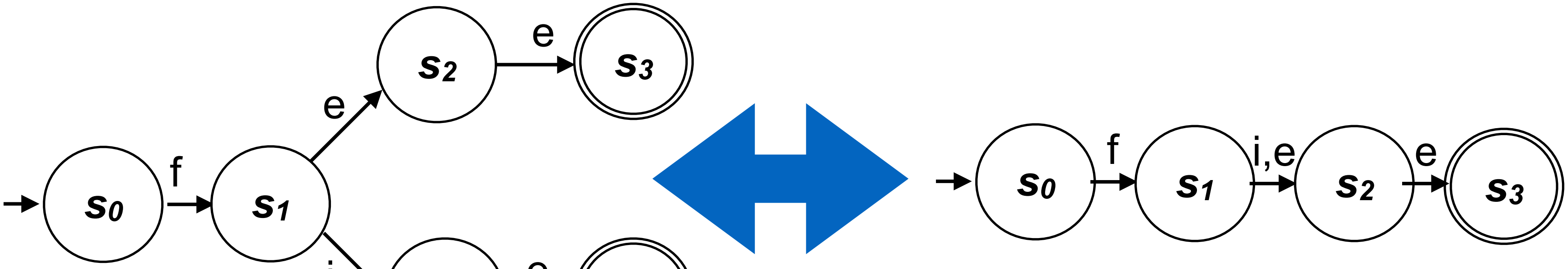
Hopcroft's algorithm



3. When Step 2 is no longer possible, we have arrived at the true equivalence classes

For our simple example, step 2 was never applicable, so the two partitions define the states of the minimized DFA

Hopcroft's algorithm: example



(states renumbered)

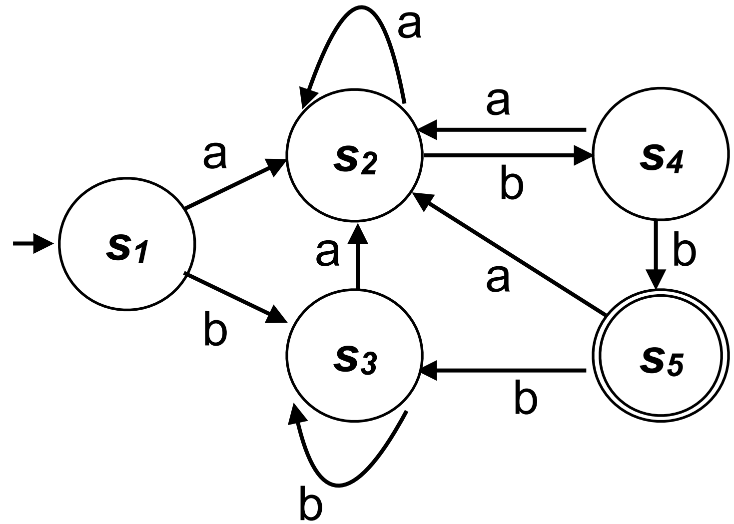
- DFA to detect **(fee | fie)**
  - s_3 and s_5 obviously (?) serve the same purpose

Step	Current Partition	Examines		
		Set	Char	Action
0	{{s3,s5},{s0,s1,s2,s4}}	–	–	–
1	{{s3,s5},{s0,s1,s2,s4}}	{s3, s5}	all	none
2	{{s3,s5},{s0,s1,s2,s4}}	{s0,s1,s2,s4}	e	split{s2,s4}
3	{{s3,s5},{s0,s1},{s2,s4}}	{s0,s1}	f	split{s1}
4	{{s3,s5},{s0},{s1},{s2,s4}}	all	all	none

More intuitive DFA minimization

Myhill-Nerode Theorem [5]
("Table Filling Method")

- Another algorithm to minimize DFAs
  (with a bit higher computational complexity than Hopcroft's)
  …but maybe easier to understand?

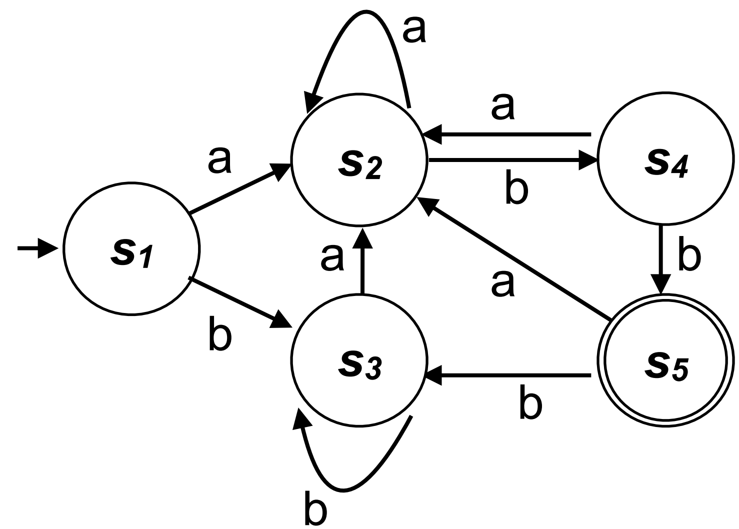


1. Draw a table for all pairs of DFA states, leave the half above (or below) the diagonal empty, including the diagonal itself
2. Mark all pairs (p, q) of states where $p \in F$ and $q \notin F$ or vice versa (here: all pairs where p or q = s_5)
   ⇒ similar to Hopcroft's first partitioning

	s_1	s_2	s_3	s_4	s_5
s_1					
s_2					
s_3					
s_4					
s_5	✗	✗	✗	✗	

Myhill-Nerode DFA minimization #1

3. If there are any unmarked pairs (p, q) such that [δ(p, x),δ(q, x)] is marked, then mark [p, q] (here 'x' is an arbitrary input symbol) – repeat this until no more markings can be made



(s_2,s_1), x=a (s_2,a) = s_2 (s_1,a) = s_2	**(s_2,s_1), x=b** (s_2,b) = s_4 (s_1,b) = s_3	**(s_3,s_1), x=a** (s_3,a) = s_2 (s_1,a) = s_2	**(s_3,s_1), x=b** (s_3,b) = s_3 (s_1,b) = s_3
(s_3,s_2), x=a (s_3,a) = s_2 (s_2,a) = s_2	**(s_3,s_2), x=b** (s_3,b) = s_3 (s_2,b) = s_4	**(s_4,s_1), x=a** (s_4,a) = s_2 (s_1,a) = s_2	**(s_4,s_1), x=b** (s_4,b) = s_5 (s_1,b) = s_2
(s_4,s_2), x=a (s_4,a) = s_2 (s_2,a) = s_2	**(s_4,s_2), x=b** (s_4,b) = s_5 (s_2,b) = s_4	**(s_4,s_3), x=a** (s_4,a) = s_2 (s_3,a) = s_2	**(s_4,s_3), x=b** (s_4,b) = s_5 (s_3,b) = s_3

✗(s_4,s_1)

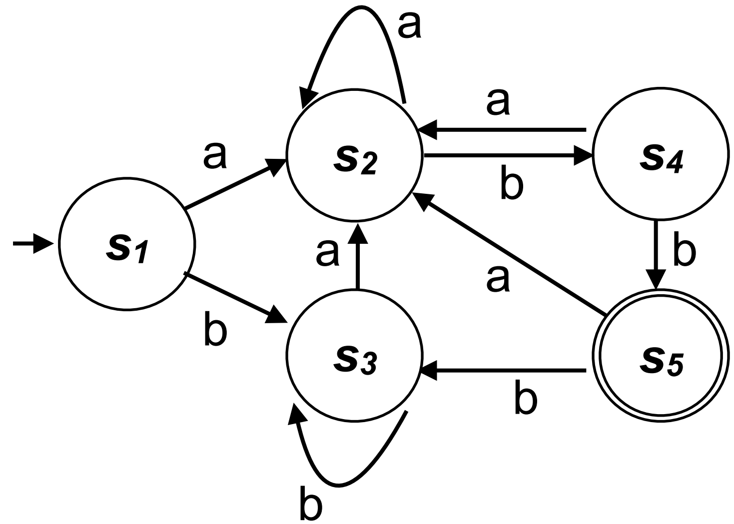
✗(s_4,s_2)

✗(s_4,s_3)

	s_1	s_2	s_3	s_4	s_5
s_1					
s_2					
s_3					
s_4	✗	✗	✗		
s_5	✗	✗	✗	✗	

Myhill-Nerode DFA minimization #2

3. If there are any unmarked pairs (p, q) such that [δ(p, x),δ(q, x)] is marked, then mark [p, q] (here ‘x’ is an arbitrary input symbol)
   – before the second iteration, only (s_2,s_1),(s_3,s_1),(s_3,s_2) are unmarked



✗(s_2,s_1)

(s_2,s_1), x=a (s_2,a) = s_2 (s_1,a) = s_2	**(s_2,s_1), x=b** (s_2,b) = s_4 (s_1,b) = s_3	**(s_3,s_1), x=a** (s_3,a) = s_2 (s_1,a) = s_2	**(s_3,s_1), x=b** (s_3,b) = s_3 (s_1,b) = s_3
(s_3,s_2), x=a (s_3,a) = s_2 (s_2,a) = s_2	**(s_3,s_2), x=b** (s_3,b) = s_3 (s_2,b) = s_4	**(s_4,s_1), x=a** (s_4,a) = s_2 (s_1,a) = s_2	**(s_4,s_1), x=b** (s_4,b) = s_5 (s_1,b) = s_2
(s_4,s_2), x=a (s_4,a) = s_2 (s_2,a) = s_2	**(s_4,s_2), x=b** (s_4,b) = s_5 (s_2,b) = s_4	**(s_4,s_3), x=a** (s_4,a) = s_2 (s_3,a) = s_2	**(s_4,s_3), x=b** (s_4,b) = s_5 (s_3,b) = s_3

✗(s_3,s_2)

✗(s_4,s_1)

✗(s_4,s_2)

✗(s_4,s_3)

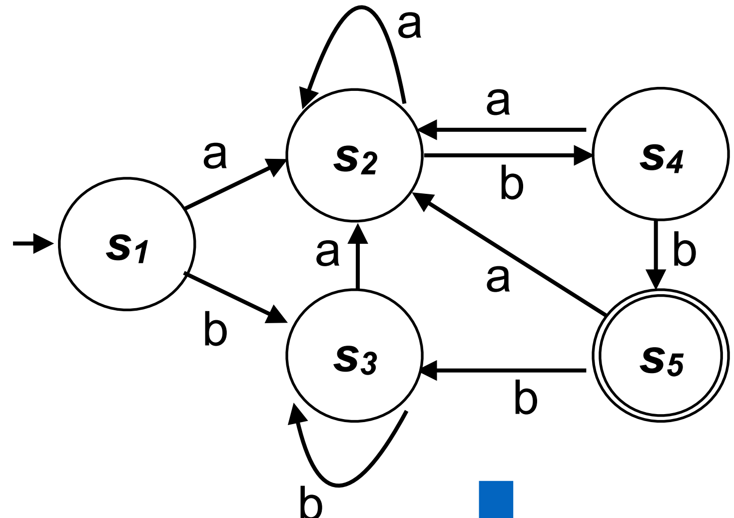
	s_1	s_2	s_3	s_4	s_5
s_1					
s_2	✗				
s_3		✗			
s_4	✗	✗	✗		
s_5	✗	✗	✗	✗	

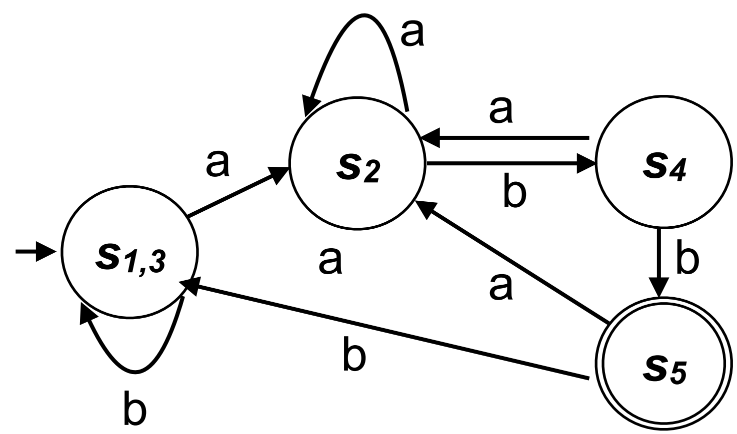
Myhill-Nerode DFA minimization

The only unmarked combination now is **(s_3,s_1)**. Both have identical subsequent states for inputs 'a' and 'b' ⇒ no marking

4. The remaining unmarked combinations of states can be combined: here, only **(s_3,s_1)** → $s_{1,3}$

	s_1	s_2	s_3	s_4	s_5
s_1					
s_2	✗				
s_3		✗			
s_4	✗	✗	✗		
s_5	✗	✗	✗	✗	

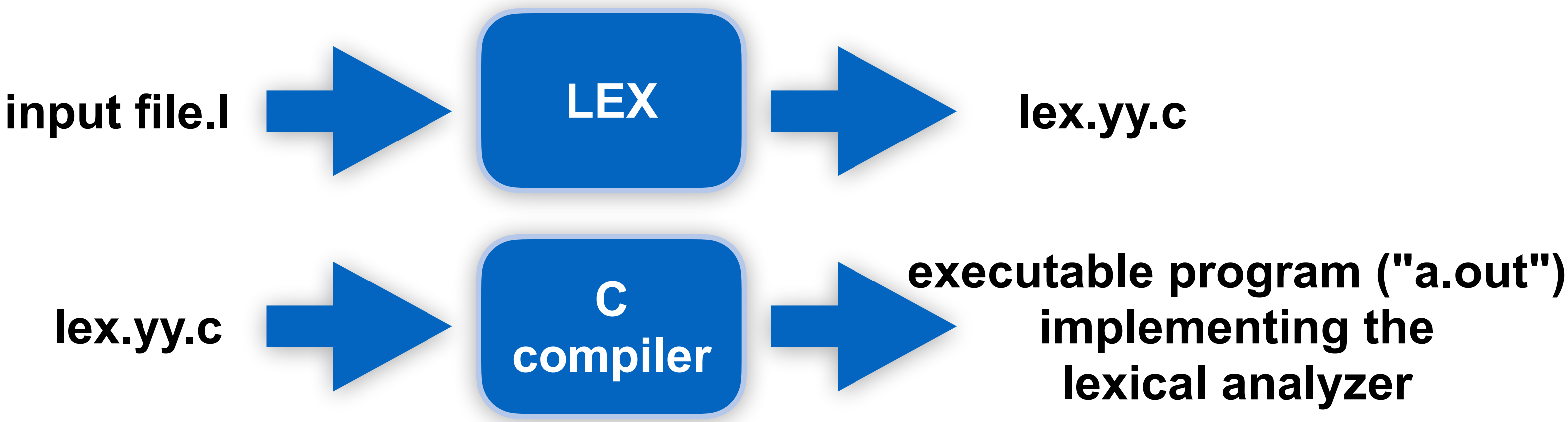




minimized DFA

A real-world scanner generator: lex

- Invented in 1975 for Unix [1]
  - today, GNU variant “flex” is still often used
- Takes a regexp-like input file and outputs a DFA implemented in C
  - using current flex: ~1700–1800 lines of code
  - using 7th edition Unix from 1979: 300 lines…
- Similar tools exist for Java (JFlex), Python (PLY), C# (C# Flex), Haskell (Alex), Eiffel (gelex), go



Lex specifications

- Lex files are suffixed *.l , and contain 3 sections:

```
<declarations>
%%
<translation rules>
%%
<functions>
```

A line containing the string
“%%"
separates the sections

- Declaration and function sections can contain regular C code that makes its way into the final product
- Translation rules are compiled into a function called yylex()
- The output is a C file

Lex declarations

```
<declarations>
%%
<translation rules>
%%
<functions>
```

- The declaration section is used to include C code (header includes, declarations of global variables or function prototypes) enclosed in “%{“ and “}%” and can also be used to add directives “% …” for lex

- The functions section is plain C code (your support function and the main function)

- The translation rules are regular expressions paired with basic blocks (actions, written as C code fragments) related to the pattern

A simple example

```
<declarations>
%%
<translation rules>
%%
<functions>
```

- A lex file that detects some regexps without any attached code:

example0.l

```
%%
[\n\t\v\ ]
if
then
endif
end
[0-9]+
%%
```

Compile with (Unix/Linux/OSX/WSL):

```
$ lex example0.l
# lex.yy.c was generated
$ ls
example0.l   lex.yy.c
# compile and link lex library
$ cc -o example0 lex.yy.c -ll
```

- This is not very useful, but it compiles…

Some action!

```
<declarations>
%%
<translation rules>
%%
<functions>
```

- We can add actions to each of the regexps:

example1.l

```
%%
[\n\t\v\ ]    { /* Do nothing, this is whitespace */ }
if            { return IF; }
then          { return THEN; }
endif         { return ENDIF; }
end           { return END; }
[0-9]+        { return INT; }
%%
```

Inside the curly brackets you write regular C code!

- We need a bit of infrastructure to make this a useful scanner

Add token definitions

```
<declarations>
%%
<translation rules>
%%
<functions>
```

- Each token is assigned a number (starting at 0 if nothing is specified):

Our scanner needs to print some output, so include the header here

example1.l

```
%{
#include <stdio.h>
enum { IF, THEN, ENDIF, INT, END };
%}
%%
[\n\t\v\ ]    { /* Do nothing, this is whitespace */ }
if            { return IF; }
then          { return THEN; }
endif         { return ENDIF; }
end           { return END; }
[0-9]+        { return INT; }
%%
```

In the declarations section you can include C code between %{ and }%. We use enums instead of #defines to automatically enumerate token numbers – failsafe!

Building a complete program

```
<declarations>
%%
<translation rules>
%%
<functions>
```

- We need a main function that repeatedly calls the generated scanner function yylex():

example1.l

```
<previous declarations>
%%
<previous regexps and actions>
%%
int main (void) {
  int token = 0;
  while (token != END) {
    token = yylex();
    switch (token) {
      case IF: printf ("Found if\n"); break;
      case THEN: printf ("Found then\n"); break;
      case ENDIF: printf ("Found endif\n"); break;
      case INT: printf ("Found integer %s\n", yytext); break;
      case END: printf ("Hanging up... bye\n"); break;
}}}
```

We call yylex() for each token

The global variable yytext contains the character string of the scanned token

Lex can run standalone

- If you need a simple scanner, you can run lex without a parser
- The example code is online, try it out!

```
$ lex example1.l
# lex.yy.c was generated
$ ls
example1.l   lex.yy.c
# compile and link lex library
$ cc -o example1 lex.yy.c -ll
# now run the scanner
$ ./example1
if 1 then 42 endif end
Found if
Found integer 1
Found then
Found integer 42
Found endif
Hanging up... bye
$
```

Type in this line and press return

Output of our scanner

Introducing states and hierarchy

- Lex enables you to define hierarchy using states
  - the states denote sub-automata
  - e.g. useful for detecting "strings inside double quotes"
- Putting the statement

```
%state STRING
```

in the declarations section declares a state named STRING

- You can then specify states in the regexps

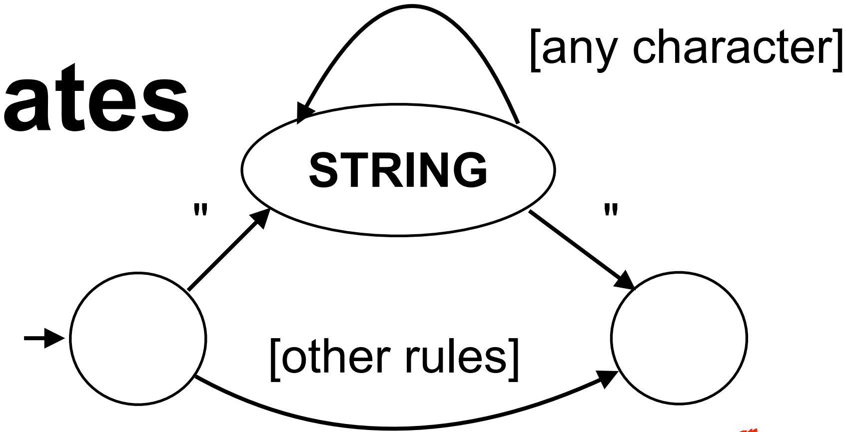
```
<INITIAL>\"
<STRING>\"
```

Double quotes need to be escaped using a \

These two specify the start and end of a string, respectively (<INITIAL> is implicitly defined)

Switching between states

- Actions allow to switch between states



Matches every second double quote

State switching

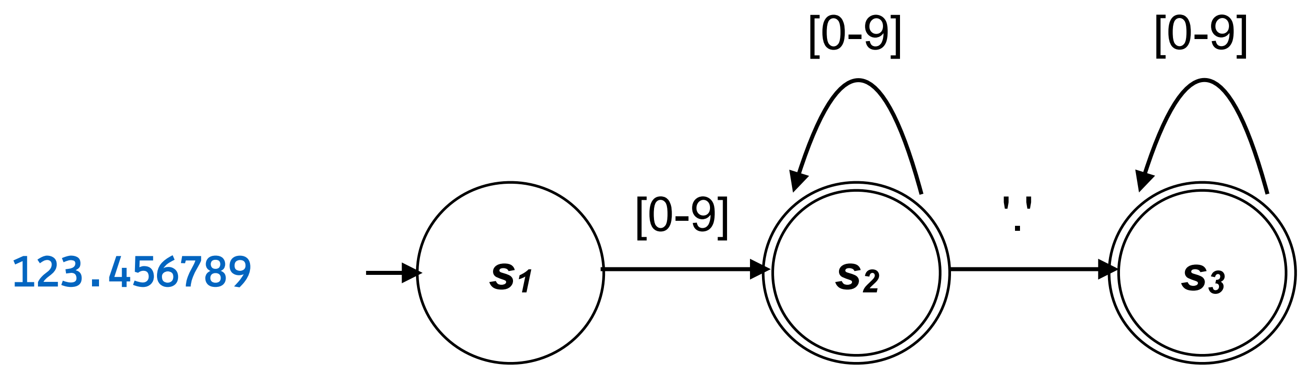
```
<INITIAL>if  { printf ( "Found 'if'\n" ); }
<INITIAL>end { printf ( "Found 'end'\n" ); return 0; }
<INITIAL>\"  { printf ( "Found string: " ); BEGIN(STRING); }
<STRING>\"   { printf ( "\n" ); BEGIN(INITIAL); }
<STRING>.    { printf ( "%c,", yytext[0] ); }
```

A dot matches arbitrary characters, the action prints the string contents

Lex matches regexps from top to bottom, so `<STRING>\"` has precedence before `<STRING>.`

Greedy automata

- When there are multiple accepting states, the DFA simulation cannot guess whether to take the first match, or continue in the hope of finding another



- Common rule it that the longest match "wins" and the input-recording buffer rolls back if input leads the DFA astray

Summary

- Lexical analysis (scanning) is required to find simple text patterns
  - expressed as a regular language
- Implementable as NFAs and DFAs
  - Equivalent representations can be constructed
- We can describe scanners as
  - graphs
  - tables
  - regular expressions (regexps)
- Scanner generators help to turn regexps into C code for a scanner

References

[1] M. E. Lesk and E. Schmidt:

Lex−A Lexical Analyzer Generator

in UNIX Programmer's Manual, Seventh Edition, Volume 2B,
Bell Laboratories Murray Hill, NJ, 1975 (the Unix standard scanner generator)

[2] Peter Bumbulis and Donald D. Cowan:

RE2C: a more versatile scanner generator

ACM Letters on Programming Languages and Systems. 2 (1–4), 1993
github.com/skvadrik/re2c/ (this one can handle Unicode input)

[3] John Hopcroft:

An n log n algorithm for minimizing states in a finite automaton

Theory of machines and computations (Proc. Internat. Sympos, Technion, Haifa), 1971,
New York: Academic Press, pp. 189–196, MR 0403320

[4] Keith Cooper and Linda Torczon:

Engineering a Compiler *(Second Edition)*

ISBN 9780120884780 (hardcover), 9780080916613 (ebook)

[5] Nerode, Anil:

Linear Automaton Transformations
Proceedings of the AMS, 9, JSTOR 2033204, 1958