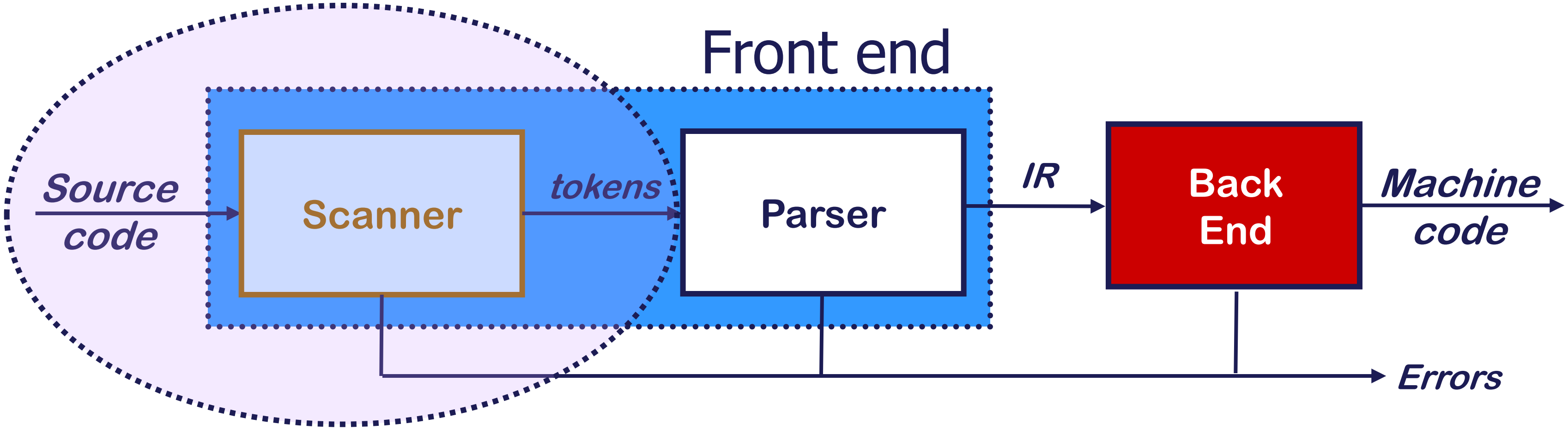
Compiler Design

Scanner

Hwansoo Han

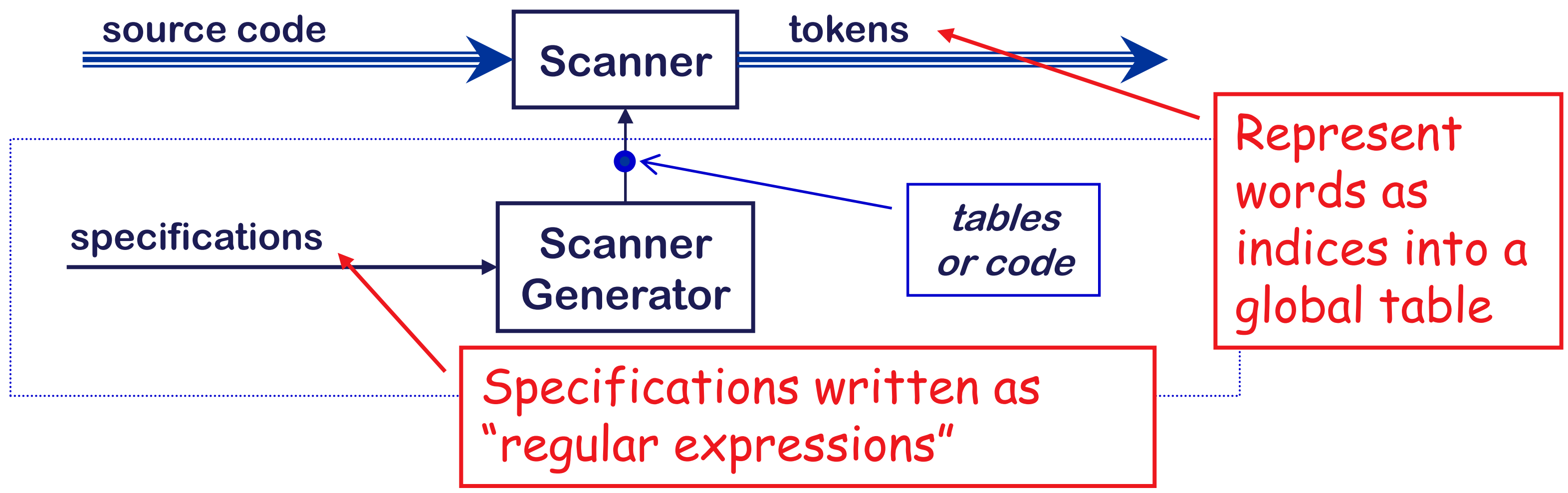
Scanner



Front end
Source code
Scanner
tokens
Parser
IR
Back End
Machine code
Errors

Scanner Generator

- **We want to avoid writing scanners by hand**
  - The scanner is the first stage in the front end
  - Specifications can be expressed using regular expressions
  - Build tables and code from a DFA



Regular Expressions

- **Regular Expression (over alphabet Σ)**
  - ε is a RE denoting the set $\{\varepsilon\}$
  - If $\underline{a}$ is in Σ, then $\underline{a}$ is a RE denoting $\{\underline{a}\}$
  - If x and y are REs denoting $L(x)$ and $L(y)$ then
    - $x \mid y$ is an RE denoting $L(x) \cup L(y)$
    - xy is an RE denoting $L(x)L(y)$
    - x^* is an RE denoting $L(x)^*$

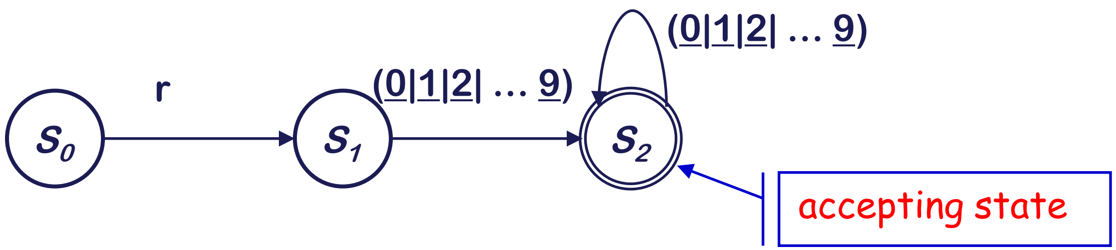
Regular Expression – Example

- **RE for recognizing register names**

  $Register \rightarrow$ r $(\underline{0}|\underline{1}|\underline{2}| \ldots | \underline{9})\ (\underline{0}|\underline{1}|\underline{2}| \ldots | \underline{9})^*$

  - Allows registers of arbitrary number
  - Requires at least one digit

- **RE corresponds to a recognizer (or DFA)**

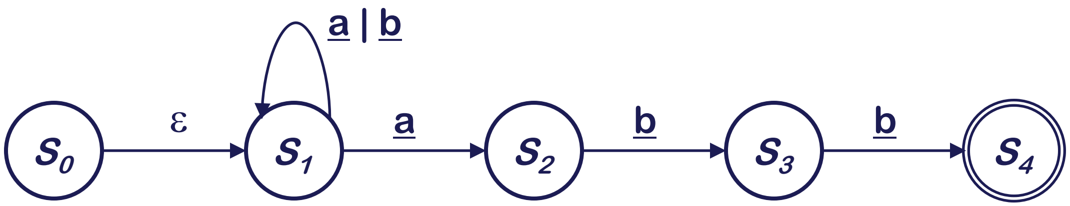


Recognizer for *Register*

Transitions on other inputs go to an error state, s_e

Non-deterministic Finite Automata (NFA)

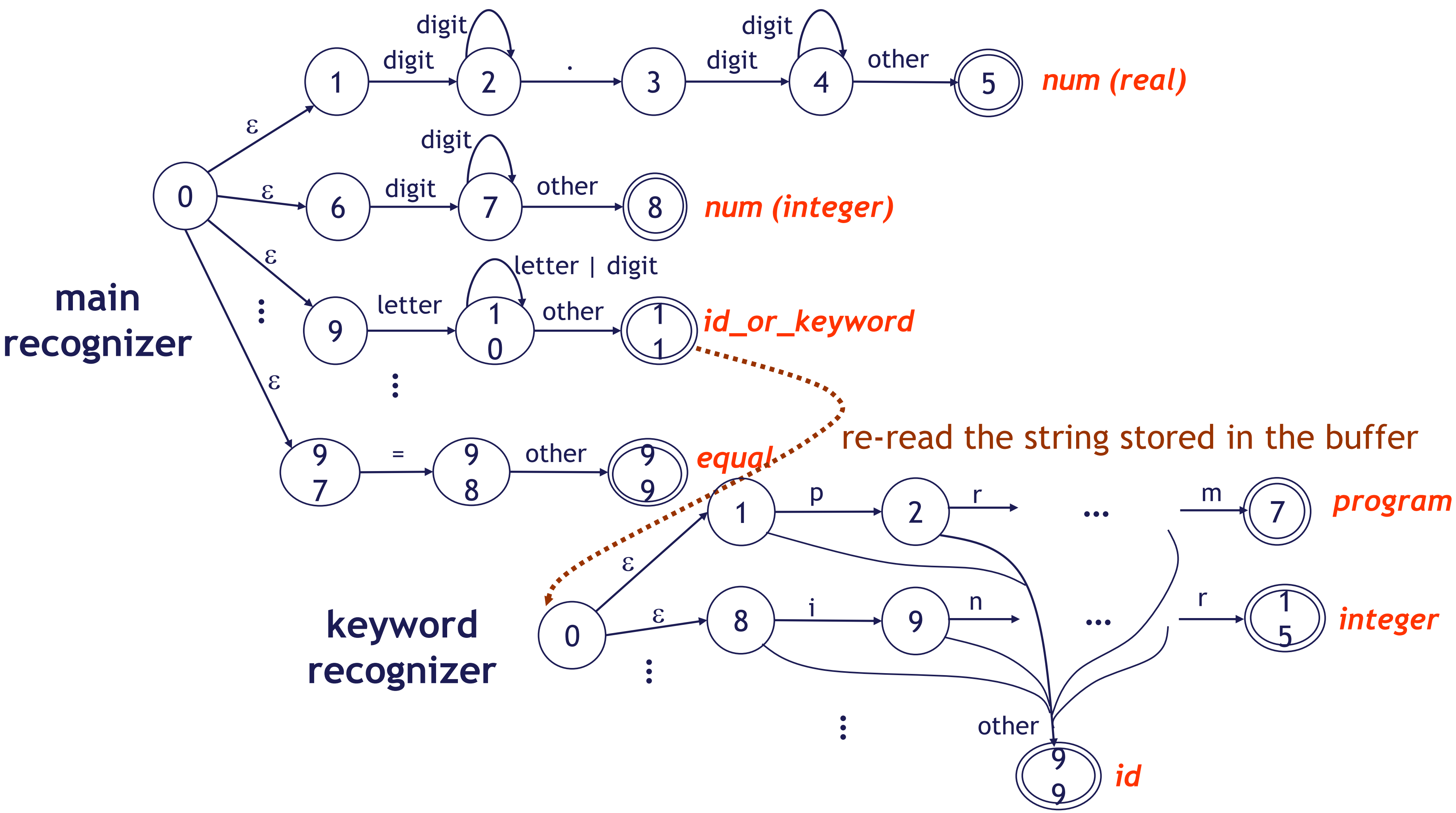
- **Each RE corresponds to a *deterministic finite automaton* (DFA)**
  - May be hard to directly construct the right DFA
  - NFA for RE such as $(\underline{a} \mid \underline{b})^* \underline{abb}$



- **NFA is a little different from DFA**
  - S_0 has a transition on ε
  - S_1 has two transitions on $\underline{a}$

Token Recognizer

- **Tokens are recognized by NFA**



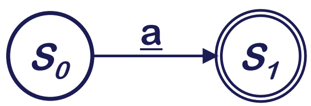
Automating Scanner Construction

- **RE→ NFA** ***(Thompson's construction)***
  - Build an NFA for each term
  - Combine them with ε-moves
- **NFA → DFA** ***(subset construction)***
  - Build the simulation
- **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

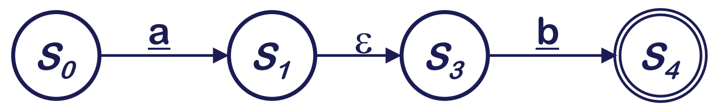
The Cycle of Constructions

RE →NFA using Thompson's Construction

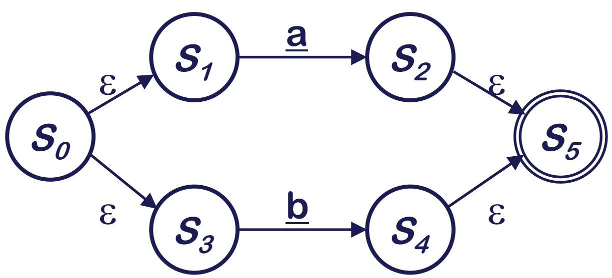
- **Key idea**
  - NFA pattern for each symbol & each operator
  - Join them with ε moves in precedence order



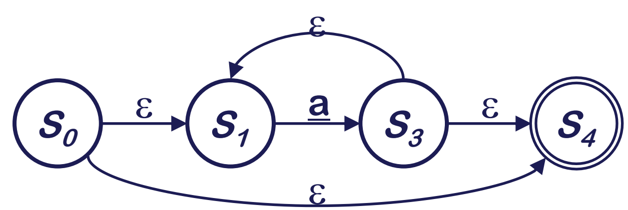
NFA for a



NFA for ab



NFA for a | b



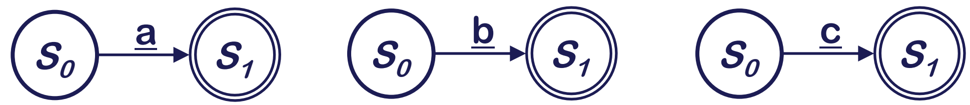
NFA for a^*

Ken Thompson, CACM, 1968

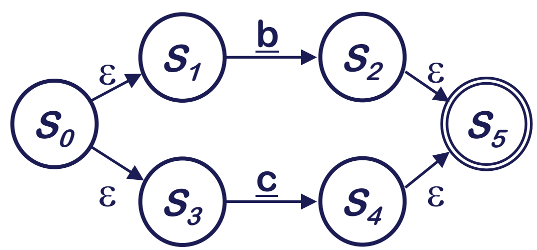
Example of Thompson's Construction

Let's try a (b | c)*

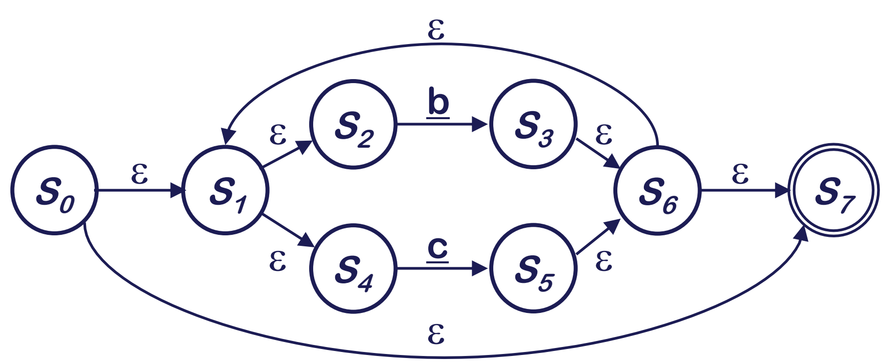
1. a, b, & c



2. b | c

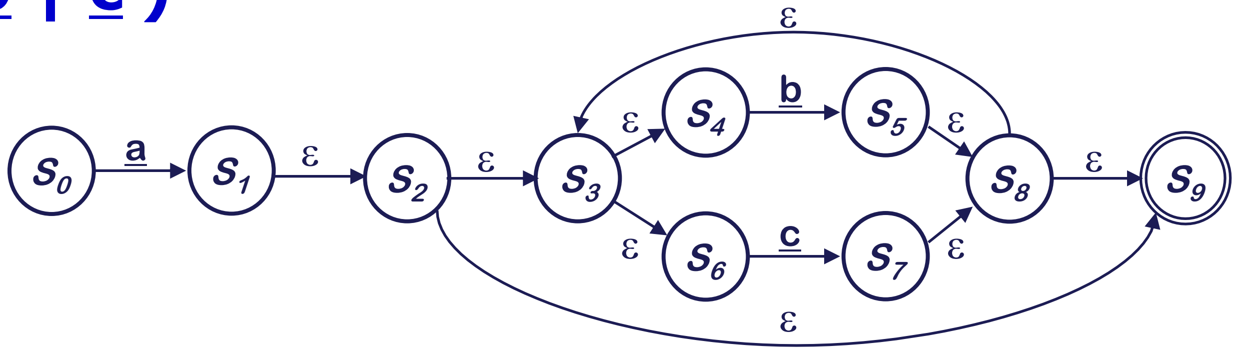


3. (b | c)*

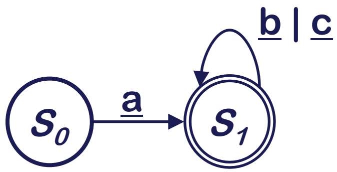


Example of Thompson's Construction (con't)

4. a (b | c)*



Of course, a human would design something simpler ...



But, we can automate production of the more complex one ...

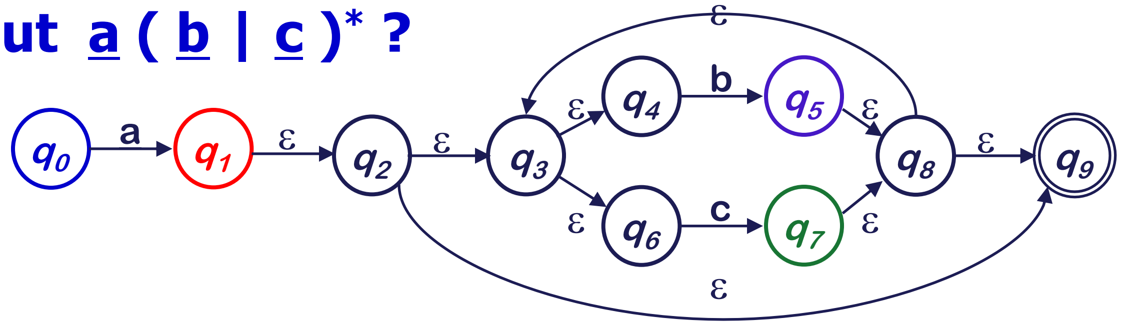
NFA $\rightarrow$DFA with Subset Construction

- **Need to build a simulation of the NFA**
- **Two key functions**
  - *Move*(s_i, $\underline{a}$) is set of states reachable from s_i by $\underline{a}$
  - *ε-closure*(s_i) is set of states reachable from s_i by ε
- **The algorithm:**
  - Start state derived from s_0 of the NFA
  - Take its ε-closure $S_0 = \varepsilon\text{-closure}(s_0)$
  - Take the image of S_0, 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…

Conversion NFA to DFA

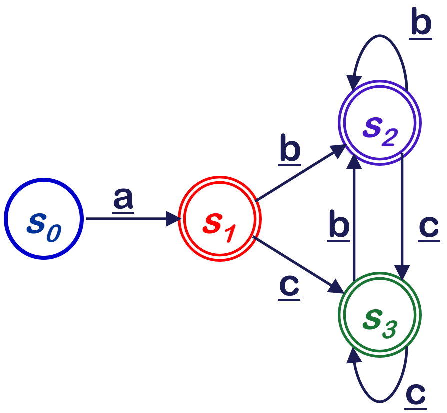
What about a (b | c)* ?



First, the subset construction: NFA → DFA

	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

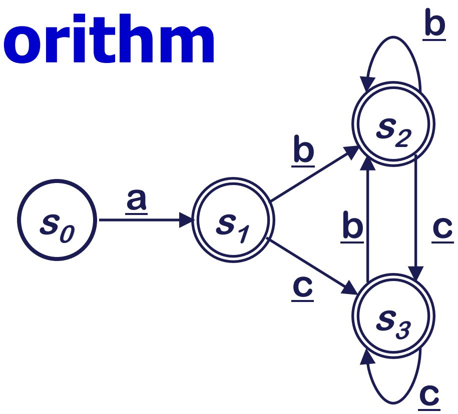


DFA Minimization

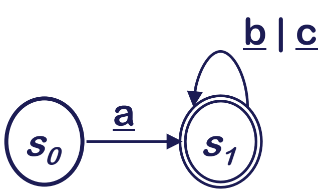
Then, apply the minimization algorithm

	Current Partition	*Split on* a	b	c
P_0	{s_1, s_2, s_3} {s_0}	*none*	*none*	*none*

final states

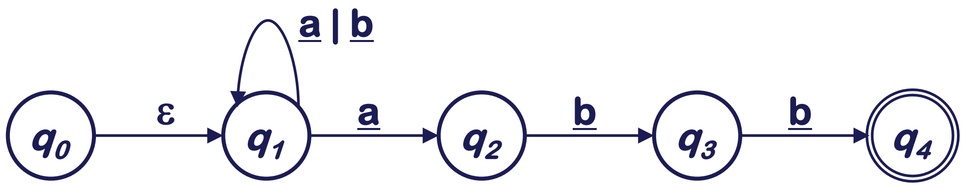


To produce the minimal DFA



Another Example

Remember (a | b)* abb ?



Our first NFA

Applying the subset construction:

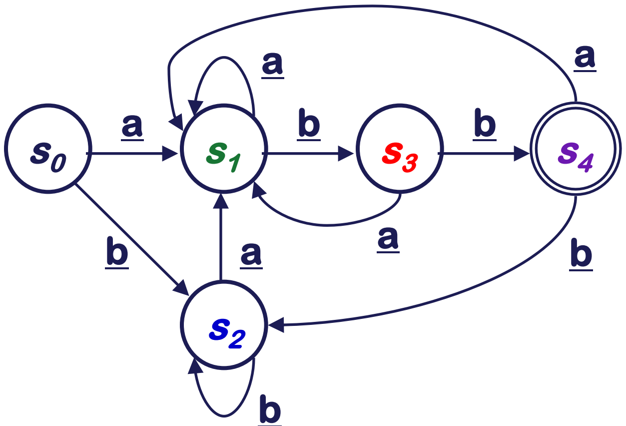
Iter.	State	Contains	ε-closure(move(s_i,a))	ε-closure(move(s_i,b))
0	s_0	q_0, q_1	q_1, q_2	q_1
1	s_1	q_1, q_2	q_1, q_2	q_1, q_3
	s_2	q_1	q_1, q_2	q_1
2	s_3	q_1, q_3	q_1, q_2	q_1, q_4
3	s_4	q_1, q_4	q_1, q_2	q_1

contains q_4 (final state)

Iteration 3 adds nothing to *S*, so the algorithm halts

Another Example (cont'd)

The DFA for (a | b)* abb



δ	a	b
s_0	s_1	s_2
s_1	s_1	s_3
s_2	s_1	s_2
s_3	s_1	s_4
s_4	s_1	s_2

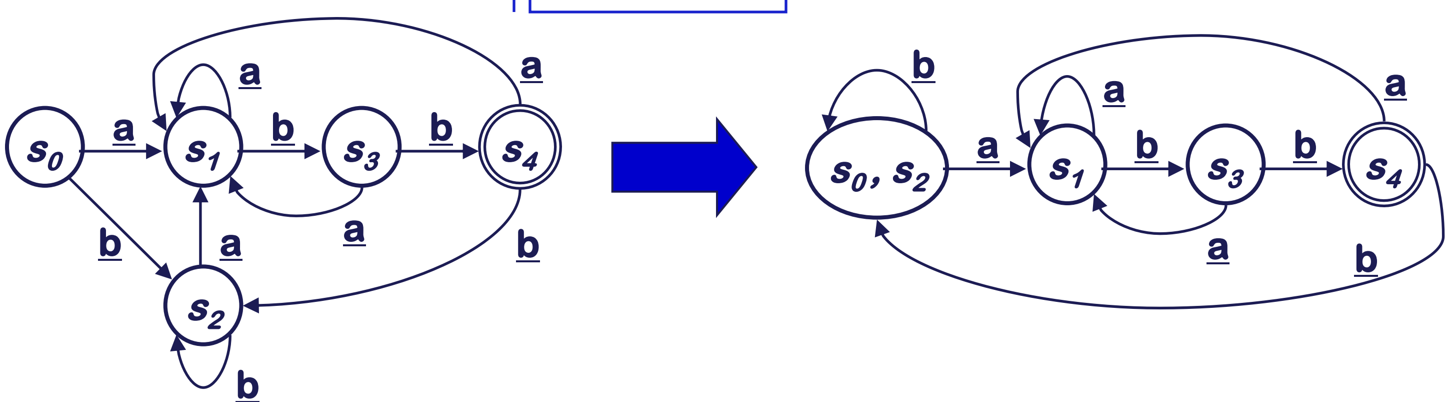
- **Not much bigger than the original**
- **All transitions are deterministic**

Another Example (cont'd)

Applying the minimization algorithm to the DFA

	Current Partition	Worklist	s	Split on a	Split on b
P_0	$\{s_4\}\ \{s_0,s_1,s_2,s_3\}$	$\{s_4\}$ $\{s_0,s_1,s_2,s_3\}$	$\{s_4\}$	none	$\{s_0, s_1, s_2\}$ $\{s_3\}$
P_1	$\{s_4\}\ \{s_3\}\ \{s_0,s_1,s_2\}$	$\{s_0,s_1,s_2\}$ $\{s_3\}$	$\{s_3\}$	none	$\{s_0, s_2\}\{s_1\}$
P_2	$\{s_4\}\ \{s_3\}\ \{s_1\}\ \{s_0,s_2\}$	$\{s_0,s_2\}\{s_1\}$	$\{s_1\}$	none	none

final state



Building Faster Scanners from the DFA

- **Table-driven recognizers waste effort**
  - Read (& classify) the next character
  - Find the next state
  - Assign to the state variable
  - Trip through case logic in *δ()* & *action()*
  - Branch back to the top
- **We can do better**
  - Encode state & actions in the code
  - Do transition tests locally
  - Generate ugly, spaghetti-like code
  - Takes (many) fewer operations per input character

```
char ← next character;
state ← s0 ;
call action(state,char);
while (char ≠ eof)
  state ← δ(state,char);
  call action(state,char);
  char ← next character;

if T(state) = final then
  report acceptance;
else
  report failure;
```

Building Faster Scanners from the DFA

- **A direct-coded recognizer for r *Digit Digit*****

```
      goto s0;
s0: word ← Ø;
      char ← next character;
      if (char = 'r')
         then goto s1;
         else goto se;
s1: word ← word + char;
      char ← next character;
      if ('0' ≤ char ≤ '9')
         then goto s2;
         else goto se;
```

```
s2: word ← word + char;
      char ← next character;
      if ('0' ≤ char ≤ '9')
         then goto s2;
         else if (char = eof)
            then report success;
            else goto se;
se: print error message;
      return failure;
```

- Many fewer operations per character
- Almost no memory operations
- Even faster with careful use of fall-through cases

Summary

- **Building scanner**
  - All this technology automates scanner construction
  - Implementer writes down the regular expressions
  - Scanner generator builds NFA, DFA, minimal DFA, and then writes out the (table-driven or direct-coded) code
  - This reliably produces fast, robust scanners

- **For most modern language features, this works**
  - You should think twice before introducing a feature that defeats a DFA-based scanner
    - insignificant blanks (Fortran: *anint* = *an int* = *an   int*)
    - non-reserved keywords (e.g. int if = 1;)