Automating Scanner Construction

RE$\rightarrow$NFA *(Thompson's construction)*

- Build an NFA for each term
- Combine them with ε-moves

NFA $\rightarrow$DFA (*subset construction*)

- Build the simulation

DFA $\rightarrow$Minimal DFA

- Hopcroft's algorithm

DFA $\rightarrow$RE

- All pairs, all paths problem
- Union together paths from s_0 to a final state

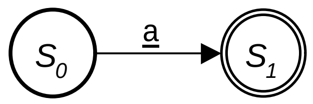
The Cycle of Constructions

$$\begin{tikzcd} \text{RE} \arrow[r] & \text{NFA} \arrow[r] & \text{DFA} \arrow[r] & \textit{minimal}\ \text{DFA} \arrow[lll, bend right] \end{tikzcd}$$

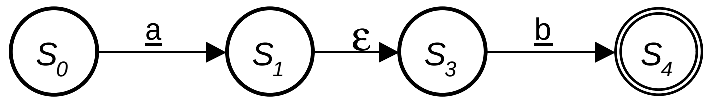
RE ®NFA using Thompson's Construction

Key idea

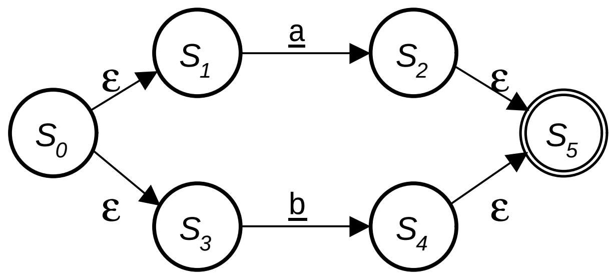
- NFA pattern for each symbol & each operator
- Join them with ε moves in precedence order



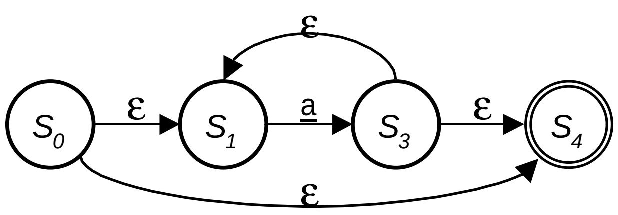
NFA for a



NFA for ab



NFA for a | b



NFA for a^*

Ken Thompson, CACM, 1968

Example of Thompson's Construction

Let's try $\underline{a}\,(\,\underline{b}\mid\underline{c}\,)^*$

1. $\underline{a}$, $\underline{b}$, & $\underline{c}$

$$\begin{tikzcd}
S_0 \arrow[r, "\underline{a}"] & S_1 & S_2 \arrow[r, "\underline{b}"] & S_3 & S_4 \arrow[r, "\underline{c}"] & S_5
\end{tikzcd}$$

2. $\underline{b}\mid\underline{c}$

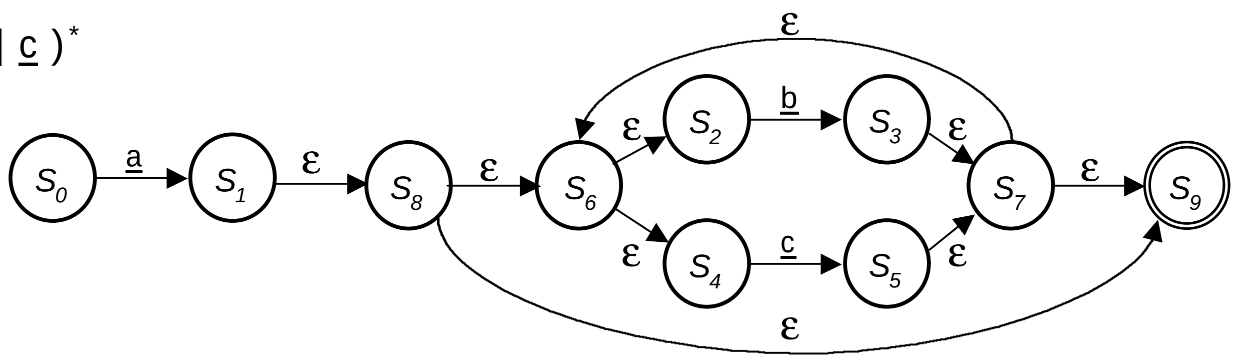
$$\begin{tikzcd}
 & S_2 \arrow[r, "\underline{b}"] & S_3 \arrow[dr] & \\
S_6 \arrow[ur] \arrow[dr] & & & S_7 \\
 & S_4 \arrow[r, "\underline{c}"] & S_5 \arrow[ur] &
\end{tikzcd}$$

3. $(\,\underline{b}\mid\underline{c}\,)^*$

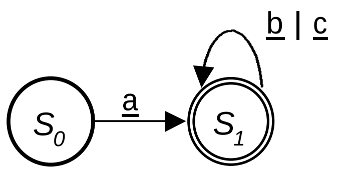
$$\begin{tikzcd}
 & & S_2 \arrow[r, "\underline{b}"] & S_3 \arrow[dr, "\varepsilon"] & & \\
S_8 \arrow[r, "\varepsilon"] \arrow[rrrrr, bend right=40, "\varepsilon"'] & S_6 \arrow[ur, "\varepsilon"] \arrow[dr, "\varepsilon"'] & & & S_7 \arrow[r, "\varepsilon"] \arrow[lll, bend right=60, "\varepsilon"'] & S_9 \\
 & & S_4 \arrow[r, "\underline{c}"] & S_5 \arrow[ur, "\varepsilon"'] & &
\end{tikzcd}$$

Example of Thompson's Construction *(continued)*

4. a (b | c)*



Of course, a human would design something simpler …



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

NFA ® DFA with Subset Construction

Need to build a simulation of the NFA

Two key functions

- *Move(s_i,<u>a</u>)* is set of states reachable by <u>a</u> from s_i
- *e-closure(s_i)* is set of states reachable by *e* from s_i

The algorithm

- Start state derived from s_0 of the NFA
- Take its ε-closure
- Work outward, trying each $\alpha \in \Sigma$ and taking its ε-closure
- Iterative algorithm that halts when the states wrap back on themselves

Sounds more complex than it is…

NFA ® DFA *with Subset Construction*

The algorithm:

```
$s_0$ ¬ e-closure($q_{0n}$)
while ( S is still changing )
  for each $s_i$ Î S
    for each a Î Σ
      $s_?$ ¬ e-closure(move($s_i$, a))
      if ( $s_?$ Ï S ) then
        add $s_?$ to S as $s_j$
        T[$s_i$, a] ¬ $s_j$
```

Let's think about why this works

The algorithm halts:

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

Þ the loop halts

S contains all the reachable NFA states

It tries each character in each s_i.

It builds every possible NFA *configuration.*

Þ *S and T form 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

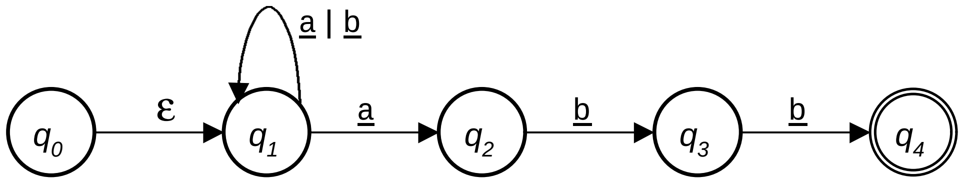
Other fixed-point computations

- Canonical construction of sets of LR(1) items
  - Quite similar to the subset construction
- Classic data-flow analysis (& Gaussian Elimination)
  - Solving sets of simultaneous set equations

We will see many more fixed-point computations

NFA ® DFA with Subset Construction

Remember (a | b)* abb ?



Applying the subset construction:

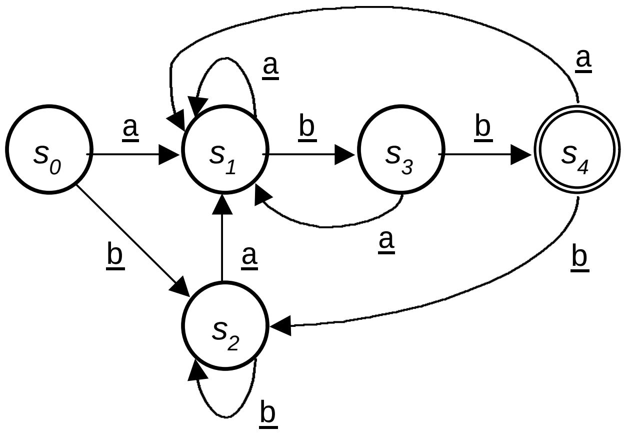
Iter.	State	Contains	ε-closure(move(s_i,a))	ε-closure(move(s_i,b))
0	s_0	q_0, q_1	q_1, q_2	q_1
1	s_1	q_1, q_2	q_1, q_2	q_1, q_3
	s_2	q_1	q_1, q_2	q_1
2	s_3	q_1, q_3	q_1, q_2	q_1, q_4
3	s_4	q_1, q_4	q_1, q_2	q_1

contains q_4 (final state)

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

NFA ® DFA with Subset Construction

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
- Use same code skeleton as before

Where are we? Why are we doing this?

RE→NFA *(Thompson's construction)*

- Build an NFA for each term
- Combine them with ε-moves

NFA →DFA *(subset construction)*

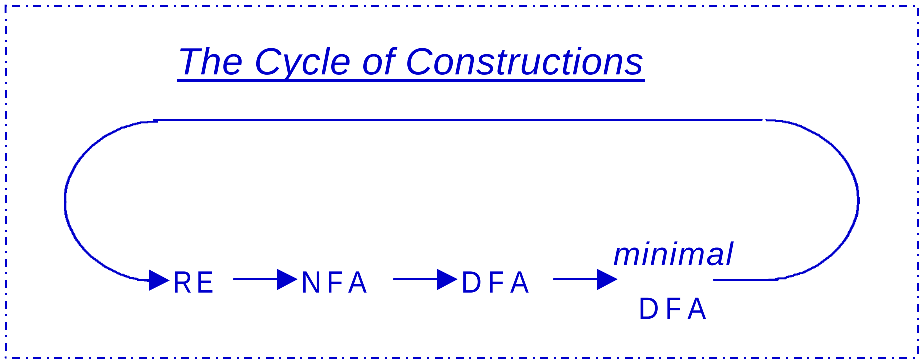
- Build the simulation

DFA →Minimal DFA

- Hopcroft's algorithm

DFA →RE

- All pairs, all paths problem
- Union together paths from s_0 to a final state



Enough theory for today

Building Faster Scanners from the DFA

Table-driven recognizers waste a lot of effort

- Read (& classify) the next character
- Find the next state
- Assign to the state variable
- Trip through case logic in *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 ¬ s_0 ;
call action(state,char);
while (char ¹ eof)
  state ¬ d(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 $s_0$;
$s_0$: word ¬ Ø;
    char  ¬ next character;
    if (char = ‘r’)
       then goto $s_1$;
       else goto $s_e$;
$s_1$: word ¬ word + char;
    char ¬ next character;
    if (‘0’   char    ‘9’)
       then goto $s_2$;
       else goto $s_e$;
s2: word ¬ word + char;
    char  ¬ next character;
    if (‘0’   char    ‘9’)
       then goto $s_2$;
       else if (char = eof)
          then report acceptance;
          else goto $s_e$;
$s_e$: print error message;
    return failure;
```

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

Building Faster Scanners

Hashing keywords versus encoding them directly

- Some compilers recognize keywords as identifiers and check them in a hash table *(some well-known compilers do this!)*
- Encoding it in the DFA is a better idea
  - > O(1) cost per transition
  - > Avoids hash lookup on each identifier

It is hard to beat a well-implemented DFA scanner