Lexical Analysis:
Constructing a Scanner from Regular Expressions

Goal

- Show how to construct a DFA to recognize any RE
- Scanner simulates a DFA to generate tokens
- Last Lecture
  - → Convert RE to an **nondeterministic finite automaton (NFA)**
    - Use Thompson's construction
- **This Lecture**
  - → Convert an NFA to a **deterministic finite automaton (DFA)**
    - Use Subset construction

Convert NFA to DFA

- NFA is a 5-tuple $(N, \Sigma, \delta_N, n_0, N_A)$
- DFA is a 5-tuple $(D, \Sigma, \delta_D, d_0, D_A)$
- Want to create a DFA that simulates the NFA

Non-trivial part is constructing D and δ_D

NFA →DFA: need to build a simulation of the NFA

Two key functions

- *Delta*(q_i , $\underline{a}$) set of states reachable from states in q_i by $\underline{a}$
  - → Returns a set of states, for each $n \in q_i$ of δ_i (n, $\underline{a}$)
- *ε-closure*(q_i) set of states reachable from q_i by ε moves

Functions help create states of DFA by removing non-deterministic edges of the NFA.

Subset Construction Algorithm in English

The algorithm:

- Start state q_0 derived from n_0 of the NFA
- Add q_0 to the Worklist

Loop while Worklist not empty

- Remove a state q from worklist
- Compute t by Delta(q, α) for each $\alpha \in \Sigma$, and take its ε-closure
- If t not in set Q

  add it to Q and Worklist

Iterate until no more states are added

Sounds more complex than it is…

The Subset Construction Algorithm

```
$q_0 \leftarrow \varepsilon$-closure($n_0$)
$Q \leftarrow \{q_0\}$
WorkList $\leftarrow \{q_0\}$
while ( WorkList is not empty )
    remove q from WorkList
    for each $\alpha \in \Sigma$
        $t \leftarrow \varepsilon$-closure(Delta($q,\alpha$))
        $T[q,\alpha] \leftarrow t$
        if ( $t \notin Q$ ) then
            add t to Q and WorkList
```

Let's think about why this works

NFA →DFA with Subset Construction

The algorithm:

```
$q_0 \leftarrow \varepsilon$-closure($n_0$)
$Q \leftarrow \{q_0\}$
WorkList $\leftarrow \{q_0\}$
while ( WorkList $\neq \phi$ )
  remove q from WorkList
  for each $\alpha \in \Sigma$
    t $\leftarrow \varepsilon$-closure(Delta(q,$\alpha$))
    T[q,$\alpha$] $\leftarrow$ t
    if ( t $\notin$ Q ) then
      add t to Q and WorkList
```

Let's think about why this works

The algorithm halts:

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

⇒ the loop halts

Q contains all the reachable NFA states

It tries each character in each q.

⇒*Q gives us D set of states of DFA*

⇒*T gives us* δ_D set of transitions of DFA

NFA →DFA with Subset Construction

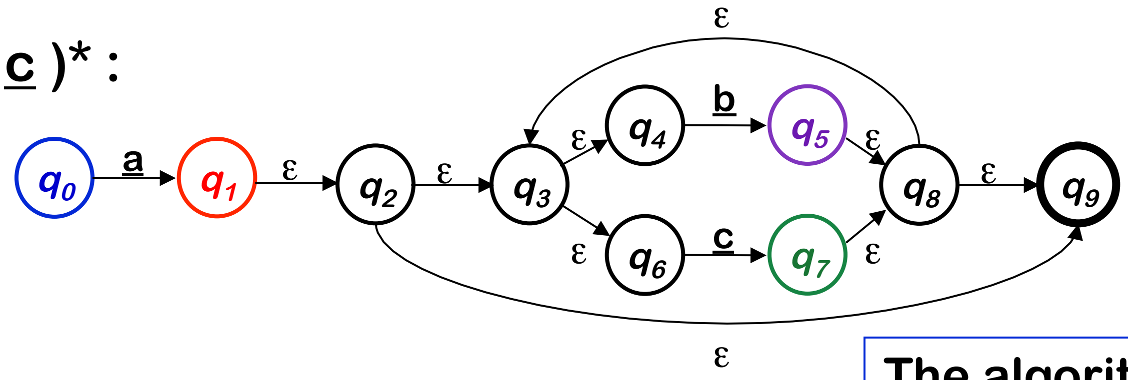
Example of a *fixed-point* computation

- Monotone construction of some finite set
- Halts when it stops adding to the set
- These computations arise in many contexts

We will see many more fixed-point computations

NFA →DFA with Subset Construction

a (b | c)* :



Applying the subset construction:

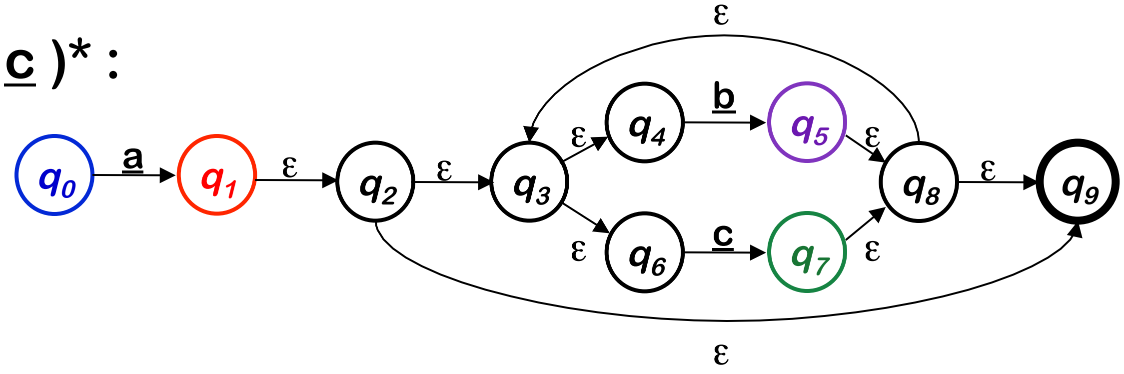
		ε-closure(Delta(q,*))		
	NFA states	a	b	c
	q_0			

The algorithm:

```
$q_0 \leftarrow \varepsilon$-closure($n_0$)
$Q \leftarrow \{q_0\}$
$WorkList \leftarrow \{q_0\}$
while ( $WorkList \neq \phi$ )
  remove q from WorkList
  for each $\alpha \in \Sigma$
      $t \leftarrow \varepsilon$-closure(Delta($q,\alpha$))
      $T[q,\alpha] \leftarrow t$
      if ( $t \notin Q$ ) then
          add t to Q and WorkList
```

NFA →DFA with Subset Construction

a (b | c)* :



Applying the subset construction:

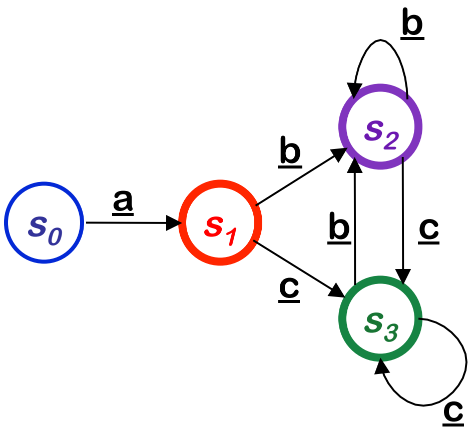
		ε-closure(Delta(q,*))		
	NFA states	a	b	c
s_0	q_0	$q_1, q_2, q_3, q_4, q_6, q_9$	none	none
s_1	$q_1, q_2, q_3, q_4, q_6, q_9$	none	$q_5, q_8, q_9, q_3, q_4, q_6$	$q_7, q_8, q_9, q_3, q_4, q_6$
s_2	$q_5, q_8, q_9, q_3, q_4, q_6$	none	s_2	s_3
s_3	$q_7, q_8, q_9, q_3, q_4, q_6$	none	s_2	s_3

Final states

NFA →DFA with Subset Construction

The DFA for a (b | c)*

- Ends up smaller than the NFA
- All transitions are deterministic
- Use same code skeleton as before



δ	a	b	c
s_0	s_1	-	-
s_1	-	s_2	s_3
s_2	-	s_2	s_3
s_3	-	s_2	s_3

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)* ✓

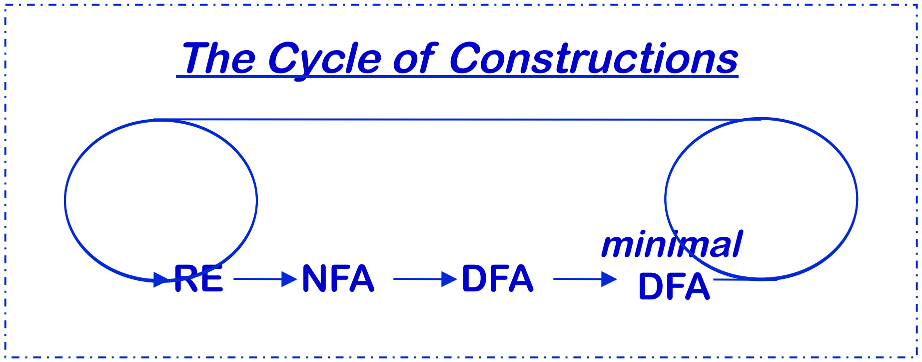
- Build the simulation

DFA → Minimal DFA

- Hopcroft's algorithm

DFA → RE

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



The Cycle of Constructions

RE → NFA → DFA → *minimal* DFA

Extra Slides

What we expect of the Scanner

- Report errors for lexicographically malformed inputs
  - → reject illegal characters, or meaningless character sequences
  - → E.g., '#' or "floop" in COOL
- Return an abstract representation of the code
  - → character sequences (e.g., "if" or "loop") turned into tokens.
- Resulting sequence of tokens will be used by the parser
- Makes the design of the parser a lot easier.

How to specify a scanner

- A scanner specification (e.g., for JLex), is list of (typically short) regular expressions.
- Each regular expressions has an action associated with it.
- Typically, an action is to return a token.
- On a given input string, the scanner will:
  - → find the longest prefix of the input string, that matches one of the regular expressions.
  - → will execute the action associated with the matching regular expression highest in the list.
- Scanner repeats this procedure for the remaining input.
- If no match can be found at some point, an error is reported.

Example of a Specification

- Consider the following scanner specification.
  1. aaa { return T1 }
  2. a*b { return T2 }
  3. b { return S }
- Given the following input string into the scanner

  aaabbaaa

  the scanner as specified above would output

  T2 T2 T1
- Note that the scanner will report an error for example on the string 'aa'.

Special Return Tokens

- Sometimes one wants to extract information out of what prefix of the input was matched.
- Example:

  “[a-zA-Z0-9]*”   { return STRING(yytext()) }
- Above RE matches every string that
  - → starts and ends with quotes, and
  - → has any number of alpha-numerical chars between them.
- Associated action returns a string token, which is the exact string that the RE matched.
- Note that yytext() will also include the quotes.
- Furthermore, note that this regular expression does not handle escaped characters.