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 \(\delta_D\)
NFA $\rightarrow$ DFA: need to build a simulation of the NFA

Two key functions

• **$\Delta(q_i, a)$** set of states reachable from states in $q_i$ by $a$
  $\rightarrow$ Returns a set of states, for each $n \in q_i$ of $\delta_i(n, a)$

• **$\varepsilon$-closure($q_i$)** set of states reachable from $q_i$ by $\varepsilon$ 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 $\text{Delta}(q, \alpha)$ for each $\alpha \in \Sigma$, and take its $\varepsilon$-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

\[
q_0 \leftarrow \varepsilon\text{-closure}(n_0)
\]

\[
Q \leftarrow \{q_0\}
\]

\[
\text{WorkList} \leftarrow \{q_0\}
\]

While (WorkList is not empty)

remove \( q \) from WorkList

for each \( \alpha \in \Sigma \)

\[
t \leftarrow \varepsilon\text{-closure}(\Delta(q, \alpha))
\]

\[
T[q, \alpha] \leftarrow t
\]

if (\( t \not\in Q \)) then

add \( t \) to \( Q \) and WorkList

Let’s think about why this works
NFA → DFA with Subset Construction

The algorithm:

\[ q_0 \leftarrow \varepsilon\text{-closure}(n_0) \]
\[ Q \leftarrow \{ q_0 \} \]
\[ \text{WorkList} \leftarrow \{ q_0 \} \]

while (WorkList ≠ φ)

remove q from WorkList

for each \( \alpha \in \Sigma \)

\[ t \leftarrow \varepsilon\text{-closure}(\text{Delta}(q, \alpha)) \]

\[ T[q, \alpha] \leftarrow t \]

if (\( t \not\in Q \)) 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 \( \delta_D \) 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 $\rightarrow$ DFA with Subset Construction

$a ( b \mid c )^*$:

Applying the subset construction:

<table>
<thead>
<tr>
<th>NFA states</th>
<th>$\varepsilon$-closure(Delta($q, \ast$))</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td></td>
</tr>
</tbody>
</table>

The algorithm:

$q_0 \leftarrow \varepsilon$-closure($n_0$)

$Q \leftarrow \{q_0\}$

WorkList $\leftarrow \{q_0\}$

while (WorkList $\neq \emptyset$)

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

\[ a(b \mid c)^* : \]

Applying the subset construction:

<table>
<thead>
<tr>
<th>NFA states</th>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_0 )</td>
<td>( q_0 )</td>
<td>( q_1, q_2, q_3, q_4, q_6, q_9 )</td>
<td>none</td>
</tr>
<tr>
<td>( s_1 )</td>
<td>( q_1, q_2, q_3, q_4, q_6, q_9 )</td>
<td>none</td>
<td>( q_5, q_8, q_9, q_3, q_4, q_6 )</td>
</tr>
<tr>
<td>( s_2 )</td>
<td>( q_5, q_8, q_9, q_3, q_4, q_6 )</td>
<td>none</td>
<td>( s_2 )</td>
</tr>
<tr>
<td>( s_3 )</td>
<td>( q_7, q_8, q_9, q_3, q_4, q_6 )</td>
<td>none</td>
<td>( s_2 )</td>
</tr>
</tbody>
</table>

Final states
The DFA for $a \ (b \mid c)^*$

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

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0$</td>
<td>$s_1$</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$s_1$</td>
<td>-</td>
<td>$s_2$</td>
<td>$s_3$</td>
</tr>
<tr>
<td>$s_2$</td>
<td>-</td>
<td>$s_2$</td>
<td>$s_3$</td>
</tr>
<tr>
<td>$s_3$</td>
<td>-</td>
<td>$s_2$</td>
<td>$s_3$</td>
</tr>
</tbody>
</table>
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
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.