Finite Automata Conversions and Lexing
Finite Automata

- Key result: all of the following have the same expressive power (i.e., they all describe regular languages):
  - Regular expressions (REs)
  - Non-deterministic finite automata (NFAs)
  - Deterministic finite automata (DFAs)

- Proof by construction
  - An algorithm exists to convert any RE to an NFA
  - An algorithm exists to convert any NFA to a DFA
  - An algorithm exists to convert any DFA to an RE
  - For every regular language, there exists a minimal DFA
    - Has the fewest number of states of all DFAs equivalent to RE
Finite Automata

- Finite automata transitions:

  Regex → NFA → DFA → Lexer

  *Kleene's construction*
  *Hopcroft's algorithm* (minimize)
  *Thompson's construction*
  *Subset construction*  
  *Brzozowski's algorithm* (direct to minimal DFA)

(dashed lines indicate transitions to a minimized DFA)
Finite Automata Conversions

- **RE to NFA**: Thompson's construction
  - Core insight: *inductively* build up NFA using “templates”
  - Core concept: use *null transitions* to build NFA quickly

- **NFA to DFA**: Subset construction
  - Core insight: DFA nodes represent *subsets* of NFA nodes
  - Core concept: use *null closure* to calculate subsets

- **DFA minimization**: Hopcroft’s algorithm
  - Core insight: create *partitions*, then keep splitting

- **DFA to RE**: Kleene's construction
  - Core insight: repeatedly eliminate states by *combining* regexes
Thompson's Construction

• Basic idea: create NFA inductively, bottom-up
  - Base case:
    • Start with individual alphabet symbols (see below)
  - Inductive case:
    • Combine by adding new states and null/epsilon transitions
    • Templates for the three basic operations
  - Invariant:
    • The NFA always has exactly one start state and one accepting state
Thompson's: Concatenation
Thompson's: Concatenation

AB
Thompson's: Union

A

B

\[ q_A \rightarrow \rightarrow f_A \]

\[ q_B \rightarrow \rightarrow f_B \]
Thompson's: Union

A|B
Thompson's: Closure
Thompson's Closure
Thompson's Construction

- **Base case**
  - transitions: $S_0 \xrightarrow{a} S_1$

- **Concatenation**
  - transitions: $q_A \rightarrow f_A \rightarrow q_B \rightarrow f_B$

- **Union**
  - transitions: $S_0 \xrightarrow{\varepsilon} q_A \xrightarrow{\varepsilon} f_A \xrightarrow{\varepsilon} S_1$

- **Closure**
  - transitions: $S_0 \xrightarrow{\varepsilon} q_A \xrightarrow{\varepsilon} f_A \xrightarrow{\varepsilon} S_1$
Subset construction

- Basic idea: create DFA incrementally
  - Each DFA state represents a subset of NFA states
  - Use null closure operation to “collapse” null/epsilon transitions
  - Null closure: all states reachable via epsilon transitions
    - Essentially: where can we go “for free?”
    - Formally: $\varepsilon$-closure$(s) = \{s\} \cup \{ t \in S \mid (s, \varepsilon \rightarrow t) \in \delta \}$
  - Simulates running all possible paths through the NFA

Null closure of $A = \{ A \}$
Null closure of $B = \{ B, D \}$
Null closure of $C = \$
Null closure of $D = \$
Subset construction

- Basic idea: create DFA incrementally
  - Each DFA state represents a subset of NFA states
  - Use **null closure** operation to “collapse” null/epsilon transitions
  - Null closure: all states reachable via epsilon transitions
    - Essentially: where can we go “for free?”
    - Formally: $\epsilon$-closure$(s) = \{s\} \cup \{ t \in S \mid (s,\epsilon \rightarrow t) \in \delta \}$
  - Simulates running all possible paths through the NFA

Null closure of A = \{ A \}
Null closure of B = \{ B, D \}
Null closure of C = \{ C, D \}
Null closure of D = \{ D \}
Subset construction

- Basic idea: create DFA incrementally
  - Each DFA state represents a subset of NFA states
  - Use null closure operation to “collapse” null/epsilon transitions
  - Null closure: all states reachable via epsilon transitions
    - Essentially: where can we go “for free?”
    - Formally: $\varepsilon$-closure$(s) = \{s\} \cup \{ t \in S \mid (s,\varepsilon \rightarrow t) \in \delta \}$
  - Simulates running all possible paths through the NFA

Null closure of $A = \{ A \}$
Null closure of $B = \{ B, D \}$
Null closure of $C = \{ C, D \}$
Null closure of $D = \{ D \}$
SubsetConstruction($S$, $\Sigma$, $s_0$, $S_A$, $\delta$):

$t_0 := \varepsilon$-closure($s_0$)

$S' := \{ t_0 \} \quad S'_A := \emptyset \quad W := \{ t_0 \}$

while $W \neq \emptyset$:

choose $u$ in $W$ and remove it from $W$

for each $c$ in $\Sigma$:

$t := \varepsilon$-closure($\delta(u,c)$)

$\delta'(u,c) = t$

if $t$ is not in $S'$ then

add $t$ to $S'$ and $W$

add $t$ to $S'_A$ if any state in $t$ is also in $S_A$

return ($S'$, $\Sigma$, $t_0$, $S'_A$, $\delta'$)
Subset Example
Subset Example
Subset Example

A -> B
B -> D
C

{A} -> {B, D}
{C, D}
SubsetExample

SubsetConstruction(S, Σ, s₀, S_A, δ):

t₀ := ε-closure(s₀)
S' := { t₀ }  S'ₐ := ∅  W := { t₀ }

while W ≠ ∅:
  choose u in W and remove it from W
  for each c in Σ:
    t := ε-closure(δ(u,c))
    δ'(u,c) = t
    if t is not in S' then
      add t to S' and W
      add t to S'ₐ if there exists a state v in t that is also in S_A

return (S', Σ, t₀, S'ₐ, δ')
Subset Example
Algorithms

• Subset construction is a fixed-point algorithm
  – Textbook: “Iterated application of a monotone function”
  – Basically: A loop that is mathematically guaranteed to terminate at some point
  – When it terminates, some desirable property holds
    • In the case of subset construction: the NFA has been converted to a DFA!
Hopcroft’s DFA Minimization

- Split into two partitions (final & non-final)
- Keep splitting a partition while there are states with **differing behaviors**
  - Two states transition to differing partitions on the same symbol
  - Or one state transitions on a symbol and another doesn’t
- When done, each partition becomes a single state
Hopcroft’s DFA Minimization

- Split into two partitions (final & non-final)
- Keep splitting a partition while there are states with **differing behaviors**
  - Two states transition to differing partitions on the same symbol
  - Or one state transitions on a symbol and another doesn’t
- When done, each partition becomes a single state
Kleene's Construction

• Replace edge labels with REs
  - "a" → "a" and "a,b" → "a|b"

• Eliminate states by combining REs
  - See pattern below; apply pairwise around each state to be eliminated
  - Repeat until only one or two states remain

• Build final RE
  - One state with "A" self-loop → "A*"
  - Two states: see pattern below

Eliminating states:

Combining final two states:
Brzozowski’s Algorithm

- Direct NFA → minimal DFA conversion
- Sub-procedures:
  - Reverse(n): invert all transitions in NFA n, adding a new start state connected to all old final states
  - Subset(n): apply subset construction to NFA n
  - Reach(n): remove any part of NFA n unreachable from start state
- Apply them all in order three times to get minimal DFA
  - First time eliminates duplicate suffixes
  - Second time eliminates duplicate prefixes
  - MinDFA(n) = Reach(Subset(Reverse(Reach(Subset(Reverse(n))))))
  - Potentially easier to code than Hopcroft’s algorithm
Brzozowski’s Algorithm

- \( \text{MinDFA}(n) = \text{Reach}(\text{Subset}(\text{Reverse}(\text{Reach}(\text{Subset}(\text{Reverse}(n))))))) \)

Example from EAC (p.76)
NFA/DFA complexity

- What are the time and space requirements to...
  - Build an NFA?
  - Run an NFA?
  - Build a DFA?
  - Run a DFA?

\[ \varepsilon \{A\} \]

\[ \{B,D\} \]

\[ \{C,D\} \]

\[ a | b \]

\[ aa^* | b \]
NFA/DFA complexity

• Thompson's construction
  - At most two new states and four transitions per regex character
  - Thus, a linear space increase with respect to the # of regex characters
  - Constant # of operations per increase means linear time as well

• NFA execution
  - Proportional to both NFA size and input string size
  - Must track multiple simultaneous “current” states

• Subset construction
  - Potential exponential state space explosion
  - A \( n \)-state NFA could require up to \( 2^n \) DFA states
  - However, this rarely happens in practice

• DFAs execution
  - Proportional to input string size only (only track a single “current” state)
NFA/DFA complexity

• NFAs build quicker (linear) but run slower
  - Better if you will only run the FA a few times
  - Or if you need features that are difficult to implement with DFAs
• DFAs build slower but run faster (linear)
  - Better if you will run the FA many times

<table>
<thead>
<tr>
<th></th>
<th>NFA</th>
<th>DFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build time</td>
<td>$O(m)$</td>
<td>$O(2^m)$</td>
</tr>
<tr>
<td>Run time</td>
<td>$O(m \times n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

$m = \text{length of regular expression}$
$n = \text{length of input string}$
Lexers

• Auto-generated
  – Table-driven: generic scanner, auto-generated tables
  – Direct-coded: hard-code transitions using jumps
  – Common tools: lex/flex and similar

• Hand-coded
  – Better I/O performance (i.e., buffering)
  – More efficient interfacing w/ other phases
  – This is what we’ll do for P2
Handling Keywords

- Issue: keywords are valid identifiers
- Option 1: Embed into NFA/DFA
  - Separate regex for keywords
  - Easier/faster for generated scanners
- Option 2: Use lookup table
  - Scan as identifier then check for a keyword
  - Easier for hand-coded scanners
  - (Thus, this is probably easier for P2)
Handling Whitespace

- Issue: whitespace is usually ignored
  - Write a regex and remove it before each new token
- Side effect: some results are counterintuitive
  - Is this a valid token? “3abc”
  - For now, it’s actually two!
  - We’ll reject them later, in the parsing phase