Lexical Analysis, III

Ignore § 2.4.4 in EaC2e. Read the replacement section posted on the course web site.

Copyright 2017, Keith D. Cooper & Linda Torczon, all rights reserved.
Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.
Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Chapter 2 in EaC2e
The Plan for Scanner Construction

RE $\rightarrow$ NFA  *(Thompson’s construction)* ✔
- Build an NFA for each term in the RE
- Combine them in patterns that model the operators

NFA $\rightarrow$ DFA *(Subset construction)* ✔
- Build a DFA that simulates the NFA

DFA $\rightarrow$ Minimal DFA
- Hopcroft’s algorithm
- Brzozowski’s algorithm

Minimal DFA $\rightarrow$ Scanner
- See § 2.5 in EaC2e

DFA $\rightarrow$ RE
- All pairs, all paths problem
- Union together paths from $s_0$ to a final state

---

**The Cycle of Constructions**
DFA Minimization

The Big Picture

• Discover sets of behaviorally equivalent states in the DFA
• Represent each such set with a single new state

Two states $s_i$ and $s_j$ are **behaviorally equivalent if and only if**:

• $\forall c \in \Sigma$, transitions from $s_i$ & $s_j$ on $c$ lead to equivalent states
• The set of paths leading from $s_i$ & $s_j$ are equivalent

A partition $P$ of a set $S$:

• A collection of subsets of $P$ such that each state $s$ is in exactly one $p_i \in P$
• The algorithm iteratively constructs partitions of the DFA’s set of states

We want a partition $P = \{ p_0, p_1, p_2, \ldots, p_n \}$ of $D$ that has two properties:

1. If $d_i$ & $d_j \in p_s$ and $c$ takes $d_i \rightarrow d_x$ and $d_j \rightarrow d_y$, then $d_x$ & $d_y \in p_t$, $\forall c, i, j, s, t$
2. If $d_i$ & $d_j \in p_s$ and $d_i \in F$ then $d_j \in F$

$D$ is the set of states for the DFA: $(D, \Sigma, \delta, s_0, D_A)$
DFA Minimization

Details of the algorithm

• Group states into maximally-sized initial sets, \textit{optimistically} \hfill (property 2)
• Iteratively subdivide those sets, based on transition graph \hfill (property 1)
• States that remain grouped together are equivalent

Initial partition: \( P_0 \) has two sets: \( \{ D_A \} \) & \( \{ D - D_A \} \)

\( D = (D, \Sigma, \delta, s_0, D_A) \)

Property 1 provides the basis for refining, or splitting, the sets

• Assume \( s_i \) & \( s_j \in p_s \), and \( \delta(s_i, a) = s_x \), & \( \delta(s_j, a) = s_y \)
• If \( s_x \) & \( s_y \) are not in the same set \( p_t \), then \( p_s \) must be split
  – \textbf{COROLLARY}: \( s_i \) has transition on \( a \), \( s_j \) does not \( \Rightarrow a \) splits \( p_s \)
• A single state in a \textbf{DFA} cannot have two transitions on \( a \)
  – Each \( p_s \) will become a \textbf{DFA} state

COMP 412, Fall 2017
DFA Minimization Algorithm (Worklist version)

\[\text{Worklist} \leftarrow \{ D_A, \{ D - D_A \} \} \]
\[\text{Partition} \leftarrow \{ D_A, \{ D - D_A \} \} \]

While (Worklist \(\neq\) \(\emptyset\)) do

  select a set \(S\) from Worklist and remove it

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

    \[\text{Image} \leftarrow \{ x \mid \delta(x, \alpha) \in S \} \]

    for each \(q \in \text{Partition}\) do

      \(p_1 \leftarrow q \cap \text{Image}\)
      \(p_2 \leftarrow q - p_1\)

      if \(p_1 \neq \emptyset\) and \(p_2 \neq \emptyset\) then
        remove \(q\) from Partition
        \[\text{Partition} \leftarrow \text{Partition} \cup p_1 \cup p_2\]

      if \(q \in \text{Worklist}\) then
        remove \(q\) from Worklist
        \[\text{Worklist} \leftarrow \text{Worklist} \cup p_1 \cup p_2\]

      else if \(|p_1| \leq |p_2|\) then
        Worklist \(\leftarrow\) Worklist \(\cup p_1\)
      else Worklist \(\leftarrow\) Worklist \(\cup p_2\)

Image is the set of states that have a transition into \(S\) on \(\alpha\): \(\delta^{-1}(S, \alpha)\)

\(p_1\) is the subset of \(q\) that transitions to \(S\) on \(\alpha\)

\(p_2\) is the rest of \(q\)

“split q”

adjust Worklist
Key Idea: Splitting Q Around Transitions on $\alpha$

Partitioning Q around S

As the algorithm considers $s$ and $\alpha$, it will split $q$.

Assume that $q$, $r$, $s$, & $t$ are sets in the current approximation to the final partition $q$ has transitions on $\alpha$ to $r$, $s$, & $t$, so it must split around $\alpha$. 
Key Idea: Splitting q around s and $\alpha$

Find maximal subset of q ($p_1$) that has an $\alpha$-transition into s

Think of $p_1$ as the image of $s$ into q under the inverse of the transition function:

$$p_1 \leftarrow \delta^{-1}(s, \alpha) \cap q$$

$p_2 = q - p_1$ must have an $\alpha$-transition to one or more other states in one or more other partitions (e.g., r & s), or states with no $\alpha$-transitions.

Otherwise, q does not split!
DFA Minimization Algorithm (Worklist version)

\[
\text{Worklist} \leftarrow \{ D_A, \{ D - D_A \} \}
\]
\[
\text{Partition} \leftarrow \{ D_A, \{ D - D_A \} \}
\]

While (Worklist \(\neq \emptyset\)) do

select a set \(S\) from Worklist and remove it

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

\[
\text{Image} \leftarrow \{ x \mid \delta(x, \alpha) \in S \}
\]

for each \(q \in \text{Partition} \) do

\[
p_1 \leftarrow q \cap \text{Image}
\]

\[
p_2 \leftarrow q - p_1
\]

if \(p_1 \neq \emptyset\) and \(p_2 \neq \emptyset\) then

remove \(q\) from Partition

\[
\text{Partition} \leftarrow \text{Partition} \cup p_1 \cup p_2
\]

if \(q \in \text{Worklist}\) then

remove \(q\) from Worklist

\[
\text{Worklist} \leftarrow \text{Worklist} \cup p_1 \cup p_2
\]

else if \(|p_1| \leq |p_2|\) then

\[
\text{Worklist} \leftarrow \text{Worklist} \cup p_1
\]

else \[
\text{Worklist} \leftarrow \text{Worklist} \cup p_2
\]

Projection is the set of states that have a transition into \(S\) on \(\alpha\):

\[
\delta^{-1}(S, \alpha)
\]

\(p_1\) is the subset of \(q\) that transitions to \(S\) on \(\alpha\)

\(p_2\) is the rest of \(q\)

And, as an implementation nit, if we just split \(S\) — that is, \(S\) was \(q\) & it split — we need a new \(S\)

"split q"

adjust Worklist

COMP 412, Fall 2017
**DFA Minimization Algorithm (Worklist version)**

\[
\text{Worklist} \leftarrow \{D_A, \{D - D_A\}\}
\]

\[
\text{Partition} \leftarrow \{D_A, \{D - D_A\}\}
\]

While (Worklist ≠ ∅) do

- select a set \(S\) from Worklist and remove it

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

    \[
    \text{Image} \leftarrow \{x | \delta(x, \alpha) \in S\}
    \]

    for each \(q \in \text{Partition}\) do

      \[
      p_1 \leftarrow q \cap \text{Image}
      p_2 \leftarrow q - p_1
      \]

      if \(p_1 \neq \emptyset\) and \(p_2 \neq \emptyset\) then

        remove \(q\) from Partition

        \[
        \text{Partition} \leftarrow \text{Partition} \cup p_1 \cup p_2
        \]

      if \(q \in \text{Worklist}\) then

        remove \(q\) from Worklist

        \[
        \text{Worklist} \leftarrow \text{Worklist} \cup p_1 \cup p_2
        \]

      else if \(|p_1| \leq |p_2|\) then

        \[
        \text{Worklist} \leftarrow \text{Worklist} \cup p_1
        \]

      else \(\text{Worklist} \leftarrow \text{Worklist} \cup p_2\)

If \(q\) is a singleton, we can skip the body of the loop because a singleton cannot split.
A Detailed Example

The DFA for (a | b)* abb

- Deterministic version of NFA from last lecture
- Specifically not the minimal DFA
- Use same code skeleton as before
A Detailed Example

Splitting a Partition

- The algorithm starts out with \{ s_0, s_1, s_2, s_3 \}, \{ s_4 \}
- How does \{ s_4 \} split \{ s_0, s_1, s_2, s_3 \}?
  - On a, no edges run from \{ s_0, s_1, s_2, s_3 \} to \{ s_4 \}, so nothing splits
A Detailed Example

Splitting a Partition

• The algorithm starts out with \{ s_0, s_1, s_2, s_3 \}, \{ s_4 \}

• How does \{ s_4 \} split \{ s_0, s_1, s_2, s_3 \}?
  – On \texttt{b}, \{ s_0, s_1, s_2, s_3 \} has edges into both \{ s_4 \} and \{ s_0, s_1, s_2, s_3 \}, so \{ s_4 \} splits \{ s_0, s_1, s_2, s_3 \} into \{ s_0, s_1, s_2 \} and \{ s_3 \}
    – \{ s_0, s_1, s_2 \} \rightarrow \{ s_0, s_1, s_2 \} on \texttt{b}
    – \{ s_3 \} \rightarrow \{ s_4 \} on \texttt{b}
A Detailed Example

Splitting a Partition

• The algorithm starts out with \( \{s_0, s_1, s_2, s_3\} \) and \( \{s_4\} \).

• How does \( \{s_4\} \) split \( \{s_0, s_1, s_2, s_3\} \)?
  - On \( b \), \( \{s_0, s_1, s_2, s_3\} \) has edges into both \( s_4 \) and \( \{s_0, s_1, s_2\} \).
  - \( \{s_0, s_1, s_2, s_3\} \) into \( \{s_0, s_1, s_2\} \) and \( \{s_3\} \).

Now, every state in \( \{s_3\} \) has the same transition on \( b \):

• Singleton set \( \Rightarrow \) same transition
• Neither \( \{s_3\} \) nor \( \{s_4\} \) can be split
• \( \{s_4\} \) causes no more splits
• \( \{s_3\} \) will split \( \{s_0, s_1, s_2\} \) into \( \{s_0, s_1\} \) and \( \{s_2\} \)

Note that when we split \( \{s_0, s_1, s_2, s_3\} \) around \( \{s_4\} \), we left behind more work — the resulting set, \( \{s_0, s_1, s_2\} \), could be split further.

In the algorithm, \( \{s_3\} \) ends up on the worklist, where it will later split \( \{s_0, s_1, s_2\} \).
### Detailed Example

<table>
<thead>
<tr>
<th>Current Partition</th>
<th>Worklist</th>
<th>s</th>
<th>Split on a</th>
<th>Split on b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{s_4} {s_0, s_1, s_2, s_3}</td>
<td>{s_4} {s_0, s_1, s_2, s_3}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example in this tabular format is for the worklist version of the algorithm.
Detailed Example

<table>
<thead>
<tr>
<th>Current Partition</th>
<th>Worklist</th>
<th>s</th>
<th>Split on a</th>
<th>Split on b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{s_4} {s_0,s_1,s_2,s_3}</td>
<td>{s_4} {s_0,s_1,s_2,s_3}</td>
<td>{s_4}</td>
<td>none</td>
</tr>
</tbody>
</table>

COMP 412, Fall 2017
## Detailed Example

<table>
<thead>
<tr>
<th>Current Partition</th>
<th>Worklist</th>
<th>s</th>
<th>Split on a</th>
<th>Split on b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{s_4} {s_0,s_1,s_2,s_3}</td>
<td>{s_4} {s_0,s_1,s_2,s_3}</td>
<td>{s_4}</td>
<td><em>none</em></td>
</tr>
</tbody>
</table>

![Diagram of a state transition graph](image)

COMP 412, Fall 2017
### Detailed Example

<table>
<thead>
<tr>
<th>Current Partition</th>
<th>Worklist</th>
<th>s</th>
<th>Split on a</th>
<th>Split on b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{s_4} {s_0, s_1, s_2, s_3}</td>
<td>{s_4} {s_0, s_1, s_2, s_3}</td>
<td>{s_4}</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td>{s_4} {s_3} {s_0, s_1, s_2}</td>
<td>{s_3} {s_0, s_1, s_2}</td>
<td></td>
<td>{s_3} {s_0, s_1, s_2}</td>
</tr>
</tbody>
</table>

**Diagram:***

\[\begin{array}{ccccccc}
 s_0 & \xrightarrow{a} & s_1 & \xrightarrow{b} & s_3 & \xrightarrow{b} & s_4 \\
 s_2 & \xrightarrow{b} & s_3 & \xrightarrow{a} & s_1 & \xrightarrow{a} & s_0 \\
\end{array}\]
## Detailed Example

<table>
<thead>
<tr>
<th>Current Partition</th>
<th>Worklist</th>
<th>$s$</th>
<th>Split on $a$</th>
<th>Split on $b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>${s_4} {s_0, s_1, s_2, s_3}$</td>
<td>${s_4}$</td>
<td>none</td>
<td>${s_3} {s_0, s_1, s_2}$</td>
</tr>
<tr>
<td>1</td>
<td>${s_4} {s_3} {s_0, s_1, s_2}$</td>
<td>${s_3}$</td>
<td>none</td>
<td>${s_3} {s_0, s_1, s_2}$</td>
</tr>
</tbody>
</table>

![Transition Diagram]

COMP 412, Fall 2017
### Detailed Example

<table>
<thead>
<tr>
<th>Current Partition</th>
<th>Worklist</th>
<th>s</th>
<th>Split on a</th>
<th>Split on b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{s_4} {s_0,s_1,s_2,s_3}</td>
<td>{s_4} {s_0,s_1,s_2,s_3}</td>
<td>{s_4}</td>
<td>none</td>
</tr>
<tr>
<td>1</td>
<td>{s_4} {s_3} {s_0,s_1,s_2}</td>
<td>{s_3} {s_0,s_1,s_2}</td>
<td>{s_3}</td>
<td>none</td>
</tr>
</tbody>
</table>

### Diagram

![Diagram](image)
## Detailed Example

<table>
<thead>
<tr>
<th>Current Partition</th>
<th>Worklist</th>
<th>s</th>
<th>Split on a</th>
<th>Split on b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>${s_4}{s_0, s_1, s_2, s_3}$</td>
<td>$s_4$</td>
<td>none</td>
<td>${s_3}{s_0, s_1, s_2}$</td>
</tr>
<tr>
<td>1</td>
<td>${s_4}{s_3}{s_0, s_1, s_2}$</td>
<td>$s_3$</td>
<td>none</td>
<td>${s_1}{s_0, s_2}$</td>
</tr>
<tr>
<td>2</td>
<td>${s_4}{s_3}{s_1}{s_0, s_2}$</td>
<td>$s_1$</td>
<td>$s_0, s_2$</td>
<td>none</td>
</tr>
</tbody>
</table>

![Diagram of state transitions](image)
### Detailed Example

<table>
<thead>
<tr>
<th>Current Partition</th>
<th>Worklist</th>
<th>s</th>
<th>Split on a</th>
<th>Split on b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{s_4}{s_0, s_1, s_2, s_3}</td>
<td>{s_4}{s_0, s_1, s_2, s_3}</td>
<td>{s_4}</td>
<td>none</td>
</tr>
<tr>
<td>1</td>
<td>{s_4}{s_3}{s_0, s_1, s_2}</td>
<td>{s_3}{s_0, s_1, s_2}</td>
<td>{s_3}</td>
<td>none</td>
</tr>
<tr>
<td>2</td>
<td>{s_4}{s_3}{s_1}{s_0, s_2}</td>
<td>{s_1}{s_0, s_2}</td>
<td>{s_1}</td>
<td>none</td>
</tr>
</tbody>
</table>

**Diagram:**

- **States:** \(s_0, s_1, s_2, s_3, s_4\)
- **Transitions:**
  - \(s_0 \xrightarrow{a} s_1\)
  - \(s_0 \xrightarrow{b} s_2\)
  - \(s_1 \xrightarrow{a} s_0, s_1, s_3\)
  - \(s_1 \xrightarrow{b} s_4\)
  - \(s_2 \xrightarrow{a} s_0, s_2\)
  - \(s_2 \xrightarrow{b} s_3, s_2\)
  - \(s_3 \xrightarrow{b} s_4\)
  - \(s_4 \xrightarrow{a} s_1\)

---

**Notes:**

- **Current Partition:** The current partition of the states.
- **Worklist:** The worklist of states to be considered next.
- **Split on a:** The state to split on if a transition occurs.
- **Split on b:** The state to split on if a transition occurs.

*COMP 412, Fall 2017*
### Detailed Example

<table>
<thead>
<tr>
<th>Current Partition</th>
<th>Worklist</th>
<th>$s$</th>
<th>Split on $a$</th>
<th>Split on $b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>${s_4} {s_0,s_1,s_2,s_3}$</td>
<td>${s_4}$</td>
<td>none</td>
<td>${s_3} {s_0,s_1,s_2}$</td>
</tr>
<tr>
<td>1</td>
<td>${s_4} {s_3} {s_0,s_1,s_2}$</td>
<td>${s_3}$</td>
<td>none</td>
<td>${s_1} {s_0,s_2}$</td>
</tr>
<tr>
<td>2</td>
<td>${s_4} {s_3} {s_1} {s_0,s_2}$</td>
<td>${s_1}$</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>3</td>
<td>${s_4} {s_3} {s_1} {s_0,s_2}$</td>
<td>${s_1} {s_0,s_2}$</td>
<td>none</td>
<td>none</td>
</tr>
</tbody>
</table>

Empty worklist $\Rightarrow$ done!

![Diagram](image-url)
### Detailed Example

<table>
<thead>
<tr>
<th>Current Partition</th>
<th>Worklist</th>
<th>s</th>
<th>Split on a</th>
<th>Split on b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{s_4} {s_0, s_1, s_2, s_3}</td>
<td>{s_4}</td>
<td>none</td>
<td>{s_3} {s_0, s_1, s_2}</td>
</tr>
<tr>
<td>1</td>
<td>{s_4} {s_3} {s_0, s_1, s_2}</td>
<td>{s_3}</td>
<td>none</td>
<td>{s_1} {s_0, s_2}</td>
</tr>
<tr>
<td>2</td>
<td>{s_4} {s_3} {s_1} {s_0, s_2}</td>
<td>{s_1}</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>3</td>
<td>{s_4} {s_3} {s_1} {s_0, s_2}</td>
<td>{s_1} {s_0, s_2}</td>
<td>none</td>
<td>none</td>
</tr>
</tbody>
</table>

20% reduction in number of states
DFA Minimization Algorithm (Worklist version)

Worklist $\leftarrow \{ D_A, \{ D - D_A \} \}$
Partition $\leftarrow \{ D_A, \{ D - D_A \} \}$

While (Worklist $\neq \emptyset$) do
    select a set $S$ from Worklist and remove it
    for each $\alpha \in \Sigma$ do
        Image $\leftarrow \{ x \mid \delta(x, \alpha) \in S \}$
        for each $q \in$ Partition do
            $p_1 \leftarrow q \cap$ Image
            $p_2 \leftarrow q - p_1$
            if $p_1 \neq \emptyset$ and $p_2 \neq \emptyset$ then
                remove $q$ from Partition
                Partition $\leftarrow$ Partition $\cup$ $p_1$ $\cup$ $p_2$
            if $q \in$ Worklist then
                remove $q$ from Worklist
                Worklist $\leftarrow$ Worklist $\cup$ $p_1$ $\cup$ $p_2$
            else if $|p_1| \leq |p_2|$ then
                Worklist $\leftarrow$ Worklist $\cup$ $p_1$
            else Worklist $\leftarrow$ Worklist $\cup$ $p_2$
    Why does this algorithm halt?
    • Fixed-point algorithm
    • DFA has finite number of states
    • Start with 2 sets in Partition
    • Splitting breaks 1 set into 2 smaller ones but never makes a set larger → Monotone behavior
    • Simple, finite limit on $|Partition|$; it cannot be $> |States|$  
    • Finite # steps, monotone increasing construction $\Rightarrow$ algorithm halts

COMP 412, Fall 2017
DFA Minimization

What about \( a (b | c)^* \)?

First, the subset construction:

<table>
<thead>
<tr>
<th>States</th>
<th>( \varepsilon)-closure(Move(s,*))</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFA</td>
<td>NFA</td>
</tr>
<tr>
<td>( s_0 )</td>
<td>( q_0 )</td>
</tr>
<tr>
<td>( s_1 )</td>
<td>( q_1 ), ( q_2 ), ( q_3 )</td>
</tr>
<tr>
<td>( s_2 )</td>
<td>( q_5 ), ( q_7 ), ( q_9 )</td>
</tr>
<tr>
<td>( s_3 )</td>
<td>( q_7 ), ( q_9 ), ( q_9 )</td>
</tr>
</tbody>
</table>

From last lecture ...
DFA Minimization

Then, apply the minimization algorithm

<table>
<thead>
<tr>
<th>Current Partition</th>
<th>Split on</th>
</tr>
</thead>
<tbody>
<tr>
<td>${s_1, s_2, s_3}, {s_0}$</td>
<td>$a$</td>
</tr>
<tr>
<td>$P_0$</td>
<td>none</td>
</tr>
</tbody>
</table>

It splits no states after the initial partition

$\Rightarrow$ The minimal DFA has two states

$\Rightarrow$ One for $\{s_0\}$

$\Rightarrow$ One for $\{s_1, s_2, s_3\}$
DFA Minimization

Then, apply the minimization algorithm

<table>
<thead>
<tr>
<th>Current Partition</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_0$</td>
<td>${s_1, s_2, s_3} \setminus {s_0}$</td>
<td>none</td>
<td>none</td>
</tr>
</tbody>
</table>

It produces this DFA

Earlier, I suggested that a human would design a simpler automaton than Thompson’s construction & the subset construction did. Minimizing that DFA produces exactly the DFA that I claimed a human would design!
Abbreviated Register Specification

Start with a regular expression

\[ r_0 | r_1 | r_2 | r_3 | r_4 | r_5 | r_6 | r_7 | r_8 | r_9 \]

Register names from zero to nine

The Cycle of Constructions
Abbreviated Register Specification

Thompson’s construction produces

The Cycle of Constructions

To make the example fit, we have eliminated some of the ε-transitions, e.g., between r and 0
Abbreviated Register Specification

Applying the subset construction yields

This is a DFA, but it has a lot of states ...

The Cycle of Constructions
Abbreviated Register Specification

Hopcroft’s algorithm

Initial sets

\[ S - F \]

\[ S_0 \rightarrow r \rightarrow S_1 \]

0 \rightarrow S_2
1 \rightarrow S_3
2 \rightarrow S_4
... 
8 \rightarrow S_{10}
9 \rightarrow S_{11}

\( F \)

\( F \) does not split.

Since no transitions leave it, there are no states to split it.

The Cycle of Constructions

Technically, this edge shows up as 10 transitions, which are combined by construction of the character classifier ...
Abbreviated Register Specification

Hopcroft’s algorithm

Initial sets

\{ S — F \} does split
Any character will split it into \{ s_0 \}, \{ s_1 \}

Technically, this edge shows up as 10 transitions, which are combined by construction of the character classifier ...
Abbreviated Register Specification

Hopcroft’s algorithm

Initial sets

{ S — F } does split
Any character will split it into { s_0 }, { s_1 }
This partition is the final partition

Technically, this edge shows up as 10 transitions, which are combined by construction of the character classifier ...
Abbreviated Register Specification

Hopcroft’s algorithm

Initial sets

\[ s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow \ldots \]

Becomes, through minimization

\[ s_0 \rightarrow s_1 \rightarrow s_f \]

Technically, this edge shows up as 10 transitions, which are combined by construction of the character classifier ...

The Critical Takeaway Points:
- The construction will build a minimal DFA
- The size of the DFA relates to the language described by the RE, not the size of the RE
- The result is a DFA, so it has \( O(1) \) cost per character
- The compiler writer can use the “most natural” or “intuitive” RE

The Cycle of Constructions

RE \( \rightarrow \) NFA \( \rightarrow \) DFA \( \xrightarrow{\text{minimal}} \) DFA
The Plan for Scanner Construction

**RE → NFA (Thompson’s construction)** ✔
- Build an NFA for each term in the RE
- Combine them in patterns that model the operators

**NFA → DFA (Subset construction)** ✔
- Build a DFA that simulates the NFA

**DFA → Minimal DFA**
- Hopcroft’s algorithm ✔
- Brzozowski’s algorithm

**Minimal DFA → Scanner**
- See § 2.5 in EaC2e

**DFA → RE**
- All pairs, all paths problem
- Union together paths from $s_0$ to a final state

---

COMP 412, Fall 2017
Brzozowski’s Algorithm for DFA Minimization

The Intuition

• The subset construction merges prefixes in the NFA

Thompson’s construction would leave ε-transitions between each single-character automaton

Subset construction eliminates ε-transitions and merges the paths for $a$. It leaves duplicate tails, such as $bc$, intact.
Brzozowski’s Algorithm

Idea: Use The Subset Construction Twice

• For an NFA $N$
  – Let $reverse(N)$ be the NFA constructed by making initial state final, adding a new start state with an $\varepsilon$-transition to each previously final state, and reversing the other edges
  – Let $subset(N)$ be the DFA produced by the subset construction on $N$
  – Let $reachable(N)$ be $N$ after removing any states that are not reachable from the initial state

• Then,

$$reachable(subset(reverse(reachable(subset(reverse(N)))))$$

is a minimal DFA that implements $N$ [Brzozowski, 1962]

Not everyone finds this result to be intuitive.
Neither algorithm dominates the other.
Brzozowski’s Algorithm

Step 1
• The subset construction on \( reverse(NFA) \) merges suffixes in original NFA

subset(reverse(NFA))
Brzozowski’s Algorithm

Step 2

• Reverse it again & use subset to merge prefixes ...

Reverse it, again

And subset it, again

The Cycle of Constructions

Minimal DFA

COMP 412, Fall 2017
Abbreviated Register Specification

Start with a regular expression
\( r_0 \mid r_1 \mid r_2 \mid r_3 \mid r_4 \mid r_5 \mid r_6 \mid r_7 \mid r_8 \mid r_9 \)

Register names from zero to nine

The Cycle of Constructions

COMP 412, Fall 2017
Abbreviated Register Specification

Thompson’s construction produces something along these lines

To make the example fit, we have eliminated some of the $\varepsilon$-transitions, e.g., between $r$ and $0$.
Abbreviated Register Specification

Applying the subset construction yields

This is a DFA, but it has a lot of states ...

The Cycle of Constructions
Abbreviated Register Specification

Applying Brzozowski’s algorithm, step 1

Technically, this edge shows up as 10 edges, which need to be combined...

The Cycle of Constructions
The Cycle of Constructions

Abbreviated Register Specification

Brzozowski, step 2 reverses that DFA and subsets it again

A skilled human might build this DFA

The Critical Point:

• The construction will build a minimal DFA
• The size of the DFA relates to the language described by the RE, not the size of the RE
• The result is a DFA, so it has $O(1)$ cost per character
• The compiler writer can use the “most natural” or “intuitive” RE

COMP 412, Fall 2017
One Last Algorithm

RE Back to DFA

Kleene’s Construction

\[
\begin{align*}
&\text{for } i \leftarrow 0 \text{ to } |D| - 1; \quad // \text{label each immediate path} \\
&\quad \text{for } j \leftarrow 0 \text{ to } |D| - 1; \\
&\quad \quad R^0_{ij} \leftarrow \{a \mid \delta(d_i, a) = d_j\}; \\
&\quad \quad \text{if } (i = j) \text{ then} \\
&\quad \quad \quad R^0_{ii} = R^0_{ii} \mid \{\varepsilon\}; \\
&\quad \text{for } k \leftarrow 0 \text{ to } |D| - 1; \quad // \text{label nontrivial paths} \\
&\quad \text{for } i \leftarrow 0 \text{ to } |D| - 1; \\
&\quad \quad \text{for } j \leftarrow 0 \text{ to } |D| - 1; \\
&\quad \quad \quad R^k_{ij} \leftarrow R^{k-1}_{ik}(R^{k-1}_{kk})^* R^{k-1}_{kj} \mid R^{k-1}_{ij} \\
&L \leftarrow \{\} \quad // \text{union labels of paths from} \\
&\text{For each final state } s_i \quad // \ s_0 \text{ to a final state } s_i \\
&L \leftarrow L \mid R^{|D| - 1}_{0i} \\
\end{align*}
\]

R^k_{ij} \text{ is the set of paths from } i \text{ to } j \text{ that include no state higher than } k

Adaptation of all points, all paths, low cost algorithm

COMP 412, Fall 2017

The Cycle of Constructions

The Wikipedia page on “Kleene’s algorithm” is pretty good. It also contains a link to Kleene’s 1956 paper. This form of the algorithm is usually attributed to McNaughton and Yamada in 1960.
Limits of Regular Languages

Not all languages are regular

\( \text{RL’s} \subset \text{CFL’s} \subset \text{CSL’s} \)

You cannot construct \text{DFA’s} to recognize these languages

- \( L = \{ p^k q^k \} \)  
  \( \text{(parenthesis languages)} \)

- \( L = \{ wcw^r \mid w \in \Sigma^* \} \)

Neither of these is a regular language \( \text{(nor an RE)} \)

But, this is a little subtle. You \text{can} construct \text{DFA’s} for

- Strings with alternating 0’s and 1’s
  \( (\varepsilon \mid 1)(01)^*(\varepsilon \mid 0) \)

- Strings with and even number of 0’s and 1’s

\text{RE’s} can count bounded sets and bounded differences
Limits of Regular Languages

Advantages of Regular Expressions

• Simple & powerful notation for specifying patterns
• Automatic construction of fast recognizers
  – \( O(1) \) cost per input character
• Many kinds of syntax can be specified with REs

Disadvantages of Regular Expressions

• Many interesting constructs are not regular
  – Balanced parentheses, nested \texttt{if}-\texttt{then} and \texttt{if}-\texttt{then}-\texttt{else} constructs
• The \texttt{DFA} recognizer has no real notion of grammatical structure
  – Gives no help with meaning