Lexical Analysis:
DFA Minimization & Wrap Up
Automating Scanner Construction

RE $\rightarrow$ NFA (Thompson’s construction)
- Build an NFA for each term
- Combine them with $\epsilon$-moves

NFA $\rightarrow$ DFA (subset construction)
- Build the simulation

DFA $\rightarrow$ Minimal DFA (today)
- Hopcroft’s algorithm

DFA $\rightarrow$ RE (not really part of scanner construction)
- All pairs, all paths problem
- Union together paths from $s_0$ to a final state
DFA Minimization

The Big Picture

• Discover sets of equivalent states
• Represent each such set with just one state
**DFA Minimization**

**The Big Picture**
- Discover sets of equivalent states
- Represent each such set with just one state

Two states are equivalent if and only if:
- The set of paths leading to them are equivalent
- $\forall \alpha \in \Sigma$, transitions on $\alpha$ lead to equivalent states \hspace{1cm} \text{(DFA)}
- $\alpha$-transitions to distinct sets $\Rightarrow$ states must be in distinct sets
DFA Minimization

The Big Picture

• Discover sets of equivalent states
• Represent each such set with just one state

Two states are equivalent if and only if:

• The set of paths leading to them are equivalent
• $\forall \alpha \in \Sigma$, transitions on $\alpha$ lead to equivalent states (DFA)
• $\alpha$-transitions to distinct sets $\Rightarrow$ states must be in distinct sets

A partition $P$ of $S$

• Each $s \in S$ is in exactly one set $p_i \in P$
• The algorithm iteratively partitions the DFA’s states
DFA Minimization

Details of the algorithm

• Group states into maximal size sets, optimistically
• Iteratively subdivide those sets, as needed
• States that remain grouped together are equivalent

Initial partition, $P_0$, has two sets: $\{F\}$ & $\{Q-F\}$  

$(D = (Q, \Sigma, \delta, q_0, F))$

Splitting a set (“partitioning a set by $a$“)

• Assume $q_a, q_b \in s$, and $\delta(q_a, a) = q_x$, & $\delta(q_b, a) = q_y$
• If $q_x$ & $q_y$ are not in the same set, then $s$ must be split
  $\rightarrow q_a$ has transition on $a$, $q_b$ does not $\Rightarrow a$ splits $s$
• One state in the final DFA cannot have two transitions on $a$
DFA Minimization

The algorithm

\[ P \leftarrow \{ F, \{Q-F\}\} \]
while ( \( P \) is still changing)
\[ T \leftarrow \{ \} \]
for each set \( S \in P \)
for each \( \alpha \in \Sigma \)
partition \( S \) by \( \alpha \)
into \( S_1 \), and \( S_2 \)
\[ T \leftarrow T \cup S_1 \cup S_2 \]
if \( T \neq P \) then
\[ P \leftarrow T \]

Why does this work?

- Partition \( P \in 2^Q \)
- Start off with 2 subsets of \( Q \)
  \( \{F\} \) and \( \{Q-F\} \)
- While loop takes \( P_i \rightarrow P_{i+1} \) by splitting 1 or more sets
- \( P_{i+1} \) is at least one step closer
to the partition with \( |Q| \) sets
- Maximum of \( |Q| \) splits

Note that

- Partitions are never combined
- Initial partition ensures that
final states are intact

This is a fixed-point algorithm!
Key Idea: Splitting S around $\alpha$

Original set $S$

The algorithm partitions $S$ around $\alpha$

$S$ has transitions on $\alpha$ to $R$, $Q$, & $T$
Key Idea: Splitting $S$ around $\alpha$

Original set $S$

$S_2$ is everything in $S - S_1$

Could we split $S_2$ further?

Yes, but it does not help asymptotically
DFA Minimization

Refining the algorithm

• As written, it examines every $S \in P$ on each iteration
  → This does a lot of unnecessary work
  → Only need to examine $S$ if some $T$, reachable from $S$, has split

• Reformulate the algorithm using a “worklist”
  → Start worklist with initial partition, $F$ and $Q-F$
  → When it splits $S$ into $S_1$ and $S_2$, place $S_2$ on worklist

This version looks at each $S \in P$ many fewer times
⇒ Well-known, widely used algorithm due to John Hopcroft
Hopcroft's Algorithm

\[ W \leftarrow \{ F, Q-F \}; \quad P \leftarrow \{ F, Q-F \}; \quad // W \text{ is the worklist, } P \text{ the current partition} \]

\[ \text{while ( } W \text{ is not empty ) do begin} \]
\[ \quad \text{select and remove } S \text{ from } W; \quad // S \text{ is a set of states} \]
\[ \quad \text{for each } \alpha \text{ in } \Sigma \text{ do begin} \]
\[ \quad \quad \text{let } I_\alpha \leftarrow \delta_\alpha^{-1}( S ); \quad // I_\alpha \text{ is set of all states that can reach } S \text{ on } \alpha \]
\[ \quad \quad \text{for each } R \text{ in } P \text{ such that } R \cap I_\alpha \text{ is not empty} \]
\[ \quad \quad \quad \text{and } R \text{ is not contained in } I_\alpha \text{ do begin} \]
\[ \quad \quad \quad \quad \text{partition } R \text{ into } R_1 \text{ and } R_2 \text{ such that } R_1 \leftarrow R \cap I_\alpha; \quad R_2 \leftarrow R - R_1; \]
\[ \quad \quad \quad \quad \text{replace } R \text{ in } P \text{ with } R_1 \text{ and } R_2; \]
\[ \quad \quad \quad \quad \text{if } R \in W \text{ then replace } R \text{ with } R_1 \text{ in } W \text{ and add } R_2 \text{ to } W; \]
\[ \quad \quad \quad \text{else if } || R_1 || \leq || R_2 || \]
\[ \quad \quad \quad \quad \quad \text{then add } R_1 \text{ to } W; \]
\[ \quad \quad \quad \text{else add } R_2 \text{ to } W; \]
\[ \quad \quad \text{end} \]
\[ \quad \text{end} \]
\[ \text{end} \]
A Detailed Example

Remember \(( a | b )^* abb ?\)  
(from last lecture)

Applying the subset construction:

<table>
<thead>
<tr>
<th>Iter.</th>
<th>State</th>
<th>Contains</th>
<th>(\varepsilon)-closure(move(s_i,a))</th>
<th>(\varepsilon)-closure(move(s_i,b))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(s_0)</td>
<td>(q_0, q_1)</td>
<td>(q_1, q_2)</td>
<td>(q_1)</td>
</tr>
<tr>
<td>1</td>
<td>(s_1)</td>
<td>(q_1, q_2)</td>
<td>(q_1, q_2)</td>
<td>(q_1, q_3)</td>
</tr>
<tr>
<td></td>
<td>(s_2)</td>
<td>(q_1)</td>
<td>(q_1, q_2)</td>
<td>(q_1)</td>
</tr>
<tr>
<td>2</td>
<td>(s_3)</td>
<td>(q_1, q_3)</td>
<td>(q_1, q_2)</td>
<td>(q_1, q_4)</td>
</tr>
<tr>
<td>3</td>
<td>(s_4)</td>
<td>(q_1, q_4)</td>
<td>(q_1, q_2)</td>
<td>(q_1)</td>
</tr>
</tbody>
</table>

Iteration 3 adds nothing to \(S\), so the algorithm halts.

Our first NFA
A Detailed Example

The DFA for \((a | b)^* \text{abb}\)

- Not much bigger than the original
- All transitions are deterministic
- Use same code skeleton as before
A Detailed Example

Applying the minimization algorithm to the DFA

<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>$P_0$</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>none</td>
<td>${s_0, s_1, s_2}$ ${s_3}$</td>
</tr>
<tr>
<td>$P_1$</td>
<td>${s_4}$ ${s_3}$ ${s_0,s_1,s_2}$</td>
<td>${s_0,s_1,s_2}$ ${s_3}$</td>
<td>none</td>
<td>${s_0, s_2} {s_1}$</td>
</tr>
<tr>
<td>$P_2$</td>
<td>${s_4}$ ${s_3}$ ${s_1}$ ${s_0,s_2}$</td>
<td>${s_0,s_2} {s_1}$</td>
<td>${s_1}$</td>
<td>none</td>
</tr>
</tbody>
</table>

Final state
DFA Minimization

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

First, 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_8 )</td>
<td>none</td>
</tr>
<tr>
<td>( s_1 )</td>
<td>( q_1, q_2, q_3, q_4, q_6, q_8 )</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>( q_5, q_8, q_9, q_3, q_4, q_6 )</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>( q_5, q_8, q_9, q_3, q_4, q_6 )</td>
</tr>
</tbody>
</table>

\( \varepsilon \)-closure(move(\( s_0 \)*))

Final states
DFA Minimization

Then, apply the minimization algorithm

<table>
<thead>
<tr>
<th>Current Partition</th>
<th>Split on</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_0$</td>
<td>${s_1, s_2, s_3} {s_0}$</td>
</tr>
</tbody>
</table>

To produce the minimal DFA

In lecture 5, we observed that a human would design a simpler automaton than Thompson’s construction & the subset construction did.

Minimizing that DFA produces the one that a human would design!
Abbreviated Register Specification

Start with a regular expression
\[ r0 \mid r1 \mid r2 \mid r3 \mid r4 \mid r5 \mid r6 \mid r7 \mid r8 \mid r9 \]

The Cycle of Constructions
Abbreviated Register Specification

Thompson’s construction produces

The Cycle of Constructions

To make it fit, we’ve eliminated the ε-transition between “r” and “0”.

minimal DFA
Abbreviated Register Specification

The subset construction builds

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

The Cycle of Constructions

RE → NFA → DFA → minimal DFA
Abbreviated Register Specification

The DFA minimization algorithm builds

This looks like what a skilled compiler writer would do!

The Cycle of Constructions

RE $\rightarrow$ NFA $\rightarrow$ DFA $\rightarrow$ minimal DFA
Limits of Regular Languages

Advantages of Regular Expressions

- Simple & powerful notation for specifying patterns
- Automatic construction of fast recognizers
- Many kinds of syntax can be specified with REs

Example — an expression grammar

\[
\begin{align*}
\text{Term} & \rightarrow [a-zA-Z] ([a-zA-z] | [0-9])^* \\
\text{Op} & \rightarrow + | - | * | / \\
\text{Expr} & \rightarrow ( \text{Term Op} )^* \text{Term}
\end{align*}
\]

Of course, this would generate a DFA ...

If REs are so useful ...

*Why not use them for everything?*
Limits of Regular Languages

Not all languages are regular

\[ \text{RL's} \subset \text{CFL's} \subset \text{CSL's} \]

You cannot construct DFA’s to recognize these languages

- \( L = \{ p^k q^k \} \) (parenthesis languages)
- \( L = \{ wcw^r \mid w \in \Sigma^* \} \)

Neither of these is a regular language (nor an RE)

But, this is a little subtle. You can construct DFA’s for

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

RE’s can count bounded sets and bounded differences
What can be so hard?

Poor language design can complicate scanning

- Reserved words are important
  
  if then then then = else; else else = then  
  
  (PL/I)

- Insignificant blanks
  
  do 10 i = 1,25  
  do 10 i = 1.25  

  (Fortran & Algol68)

- String constants with special characters
  
  newline, tab, quote, comment delimiters, ...

  (C, C++, Java, ...)

- Finite closures
  
  → Limited identifier length
  
  → Adds states to count length

  (Fortran 66 & Basic)
How does a compiler scan this?
• First pass finds & inserts blanks
• Can add extra words or tags to create a scannable language
• Second pass is normal scanner

Example due to Dr. F.K. Zadeck
Table-driven recognizers waste 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
Building Faster Scanners from the DFA

A direct-coded recognizer for \texttt{r Digit Digit*}

\begin{verbatim}
goto s_0;
s_0: word \leftarrow \emptyset;
    char \leftarrow \text{next character};
    \text{if} (\text{char} = \text{'r'})
      \text{then goto } s_1;
    \text{else goto } s_e;

s_1: word \leftarrow \text{word} + \text{char};
    char \leftarrow \text{next character};
    \text{if} ('0' \leq \text{char} \leq '9')
      \text{then goto } s_2;
    \text{else goto } s_e;

s_2: word \leftarrow \text{word} + \text{char};
    char \leftarrow \text{next character};
    \text{if} ('0' \leq \text{char} \leq '9')
      \text{then goto } s_2;
    \text{else if} (\text{char} = \text{eof})
      \text{then report success;}
      \text{else goto } s_e;

s_e: \text{print error message;}
    \text{return failure;}
\end{verbatim}

\begin{itemize}
\item Many fewer operations per character
\item Almost no memory operations
\item Even faster with careful use of fall-through cases
\end{itemize}
Building Faster Scanners

Hashing keywords versus encoding them directly

- Some *(well-known)* compilers recognize keywords as identifiers and check them in a hash table.
- Encoding keywords 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.*
Building Scanners

The point

- All this technology lets us automate scanner construction
- Implementer writes down the regular expressions
- Scanner generator builds NFA, DFA, minimal DFA, and then writes out the (table-driven or direct-coded) code
- This reliably produces fast, robust scanners

For most modern language features, this works

- You should think twice before introducing a feature that defeats a DFA-based scanner
- The ones we’ve seen (e.g., insignificant blanks, non-reserved keywords) have not proven particularly useful or long lasting
Extra Slides Start Here
Some Points of Disagreement with EAC

- **Table-driven scanners are not fast**
  - EaC doesn’t say they are slow; it says you can do better
  - Scanning is a small part of the work in a compiler, so in most cases it cannot make a large % improvement: decide where to spend your effort!

- **Faster code can be generated by embedding scanner in code**
  - This was shown for both LR-style parsers and for scanners in the 1980s; flex and its derivatives are an example

- **Hashed lookup of keywords is slow**
  - EaC doesn’t say it is slow. It says that the effort can be folded into the scanner so that it has no extra cost. Compilers like GCC use hash lookup. A word must fail in the lookup to be classified as an identifier. With collisions in the table, this can add up. At any rate, the cost is unneeded, since the DFA can do it for O(1) cost per character. But again, what % of total cost to compile is this?
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

```plaintext
char ← next character;
state ← s₀;
call action(state,char);
while (char ≠ eof)
    state ← δ(state,char);
call action(state,char);
char ← next character;
if T(state) = final then
    report acceptance;
else
    report failure;
```
Key Idea: Splitting S around $\alpha$

Find partition $I$ that has an $\alpha$-transition into $S$

This part must have an $\alpha$-transition to some other state!