Lexical Analysis — Part II: Constructing a Scanner from Regular Expressions

Copyright 2003, Keith D. Cooper, Ken Kennedy & 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.
Quick Review

Previous class:

→ The scanner is the first stage in the front end
→ Specifications can be expressed using regular expressions
→ Build tables and code from a DFA
→ Regular expressions, NFAs and DFAs
Goal

• We will show how to construct a finite state automaton to recognize any RE

• Overview:
  → Direct construction of a **nondeterministic finite automaton (NFA)** to recognize a given RE
    ▪ Requires $\varepsilon$-transitions to combine regular subexpressions
  → Construct a **deterministic finite automaton (DFA)** to simulate the NFA
    ▪ Use a set-of-states construction
  → Minimize the number of states
    ▪ Hopcroft state minimization algorithm
  → Generate the scanner code
    ▪ Additional specifications needed for details
**RE \(\rightarrow\) NFA using Thompson’s Construction**

**Key idea**
- NFA pattern for each symbol & each operator
- Join them with \(\varepsilon\) moves in precedence order

Ken Thompson, CACM, 1968
Example of Thompson’s Construction

Let’s try $a \ (b \ | \ c)^*$

1. $a$, $b$, & $c$

2. $b \ | \ c$

3. $(b \ | \ c)^*$
Example of Thompson’s Construction  (con’t)

4. \( a (b \mid c)^* \)

Of course, a human would design something simpler ...

But, we can automate production of the more complex one ...
NFA $\rightarrow$ DFA with Subset Construction

Need to build a simulation of the NFA

Two key functions

\begin{itemize}
    \item $\text{Move}(s_i, a)$ is set of states reachable from $s_i$ by $a$
    \item $\varepsilon$-closure($s_i$) is set of states reachable from $s_i$ by $\varepsilon$
\end{itemize}

The algorithm:

\begin{itemize}
    \item Start state derived from $s_0$ of the NFA
    \item Take its $\varepsilon$-closure $S_0 = \varepsilon$-closure($s_0$)
    \item Take the image of $S_0$, $\text{Move}(S_0, \alpha)$ for each $\alpha \in \Sigma$, and take its $\varepsilon$-closure
    \item Iterate until no more states are added
\end{itemize}

Sounds more complex than it is...
NFA $\rightarrow$ DFA with Subset Construction

The algorithm:

$$s_0 \leftarrow \varepsilon\text{-closure}(q_{0n})$$

while ( $S$ is still changing )

for each $s_i \in S$

for each $\alpha \in \Sigma$

$$s_? \leftarrow \varepsilon\text{-closure}(\text{Move}(s_i, \alpha))$$

if ( $s_? \not\in S$ ) then

add $s_?$ to $S$ as $s_j$

$$T[s_i, \alpha] \leftarrow s_j$$

Let’s think about why this works

The algorithm halts:

1. $S$ contains no duplicates (test before adding)
2. $2^{Q_n}$ is finite
3. while loop adds to $S$, but does not remove from $S$ (monotone)
   $\Rightarrow$ the loop halts

$S$ contains all the reachable NFA states

It tries each character in each $s_r$

It builds every possible NFA configuration.

$\Rightarrow S$ and $T$ form the DFA
NFA $\rightarrow$ DFA with Subset Construction

Example of a fixed-point computation
- Monotone construction of some finite set
- Halts when it stops adding to the set
- Proofs of halting & correctness are similar
- These computations arise in many contexts

Other fixed-point computations
- Canonical construction of sets of LR(1) items
  - Quite similar to the subset construction
- Classic data-flow analysis (& Gaussian Elimination)
  - Solving sets of simultaneous set equations

*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 (move(s,*))</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0$</td>
<td>$q_0$, $q_1$, $q_2$, $q_3$, $q_4$, $q_6$, $q_9$</td>
</tr>
<tr>
<td>$s_1$</td>
<td>$q_1$, $q_2$, $q_3$, $q_4$, $q_6$, $q_9$</td>
</tr>
<tr>
<td>$s_2$</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>
</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
Where are we? Why are we doing this?

RE $\rightarrow$ NFA \emph{(Thompson’s construction)} $\sqrt{\phantom{x}}$
- Build an NFA for each term
- Combine them with $\varepsilon$-moves

NFA $\rightarrow$ DFA \emph{(subset construction)} $\sqrt{\phantom{x}}$
- Build the simulation

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

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

\emph{The Cycle of Constructions}
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
• \( \forall \alpha \in \Sigma, \text{ transitions on } \alpha \text{ lead to equivalent states} \) \hspace{1cm} (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: $\{F\} & \{Q-F\}$

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 $\alpha$

The algorithm partitions $S$ around $\alpha$

Original set $S$

$S$ has transitions on $\alpha$ to $R$, $Q$, & $T$
Original set $S$

$S_1$

$S_2$

$S_2$ 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
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

RE → NFA → DFA → minimal DFA
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”.

RE → NFA → DFA → minimal DFA
The subset construction builds

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

The DFA minimization algorithm builds

This looks like what a skilled compiler writer would do!

The Cycle of Constructions

---

The Cycle of Constructions

---

---

---

---

---
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

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

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

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

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

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

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

- Strings with alternating 0’s and 1’s
  \( (\epsilon | 1)(01)^*(\epsilon | 0) \)

- Strings with and 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
  \[
  \text{if then then then = else; else else = then} \quad \text{(PL/I)}
  \]

• Insignificant blanks
  \[
  \begin{align*}
  \text{do 10 i = 1,25} \\
  \text{do 10 i = 1.25}
  \end{align*}
  \quad \text{(Fortran & Algol68)}
  \]

• String constants with special characters
  newline, tab, quote, comment delimiters, …
  \quad \text{(C, C++, Java, …)}

• Finite closures
  \[
  \rightarrow \text{Limited identifier length} \\
  \rightarrow \text{Adds states to count length} \quad \text{(Fortran 66 & Basic)}
  \]
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