Chapter 3: Converting Regular Expressions to NFAs

In linear time from Regular Expressions to NFAs

Berry-Sethi Approach

Berry-Sethi Algorithm
Produces exactly \( n + 1 \) states without \( \epsilon \)-transitions and demonstrates \( \rightarrow \) Equality Systems and \( \rightarrow \) Attribute Grammars

Idea:
The automaton tracks (conceptionally via a marker \( \ast \)), in the syntax tree of a regular expression, which subexpressions in \( \varphi \) are reachable consuming the rest of input \( w \).

Thompson's Algorithm
Produces \( O(n) \) states for regular expressions of length \( n \).
... for example:

\[(a|b)^*a(a|b)\]

... for example:

\[w = bbab : \]

... for example:

\[w = bbab : \]
... for example:

\[ w = baaba : \]

... for example:

\[ w = aa : \]

... for example:

\[ w = aa : \]

... for example:

\[ w = aa : \]
Berry-Sethi Approach

... for example:

\[ w = \]

In general:

- Input is only consumed by the leaves.
- Navigation in the tree is done without consuming input, i.e. via \(\epsilon\)-transition.
- For a formal construction we need identifiers for states.
- Therefore we use the subexpression, corresponding to the subtree, dominated by the particular node.
- There are possibly identical subexpressions in one regular expression.

we enumerate the leaves...
Berry-Sethi Approach

In general:
- Input is only consumed by the leaves.
- Navigation in the tree is done without consuming input, i.e. via ε-transition.
- For a formal construction we need identifiers for states.
- Therefore we use the subexpression, corresponding to the subtree, dominated by the particular node.
- There are possibly identical subexpressions in one regular expression.

⇒ we enumerate the leaves ...

Berry-Sethi Approach

... for example:

Berry-Sethi Approach (naive version)

Construction (naive version):

States: \( \cdot r, r \cdot \) with \( r \) nodes of \( \varepsilon \);
Start state: \( \cdot r \);
Final state: \( r \cdot \);
Transitions: for leaves \( r = [\text{a}] \) we require: \( [\varepsilon] \leftarrow [\varepsilon, r] \) \( r \)

The leftover transitions are:

<table>
<thead>
<tr>
<th>( r )</th>
<th>Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_1 )</td>
<td>( { \cdot r, r \cdot } )</td>
</tr>
<tr>
<td>( r_2 )</td>
<td>( { \cdot r, r \cdot } )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( r_1 \cdot r_2 )</th>
<th>Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_1 \cdot r_2 )</td>
<td>( { \cdot r, r \cdot } )</td>
</tr>
<tr>
<td>( r_1 \cdot r_2 )</td>
<td>( { \cdot r, r \cdot } )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( r_1^* )</th>
<th>Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_1^* )</td>
<td>( { \cdot r, r \cdot } )</td>
</tr>
<tr>
<td>( r_1^* )</td>
<td>( { \cdot r, r \cdot } )</td>
</tr>
</tbody>
</table>
Berry-Sethi Approach

... for example:

![Tree Diagram]

Berry-Sethi Approach

Discussion:
- Most transitions navigate through the expression
- The resulting automaton is in general **nondeterministic**

Berry-Sethi Approach (naive version)

Construction (naive version):

- **States**: \( r, r^* \) with \( r \) nodes of \( e \);
- **Start state**: \( e^* \);
- **Final state**: \( e^* \);
- **Transitions**: for leaves \( r \equiv [I] \) we require: \((e, x, r^*)\).

The leftover transitions are:

<table>
<thead>
<tr>
<th>( r )</th>
<th>Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_1 )</td>
<td>((e, e, r_1))</td>
</tr>
<tr>
<td>( r_2 )</td>
<td>((e, e, r_2))</td>
</tr>
<tr>
<td>( r_1 \cdot r_2 )</td>
<td>((e, e, r_1))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( r )</th>
<th>Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_1 )</td>
<td>((e, e, r_1))</td>
</tr>
<tr>
<td>( r_2 )</td>
<td>((e, e, r_2))</td>
</tr>
</tbody>
</table>

Berry-Sethi Approach

Discussion:
- Most transitions navigate through the expression
- The resulting automaton is in general **nondeterministic**

**Strategy for the sophisticated version:**
Avoid generating \( e \)-transitions
Berry-Sethi Approach

**Discussion:**
- Most transitions navigate through the expression
- The resulting automaton is in general **nondeterministic**

⇒ **Strategy for the sophisticated version:**
Avoid generating $\epsilon$-transitions

**Necessary node-attributes:**
- `empty` can the subexpression $r$ consume $\epsilon$?
- `first` the set of read states below $r$, which **may** be reached **first**, when descending into $r$.
- `next` the set of read states on the right of $r$, which **may** be reached **first** in the traversal **after** $r$.
- `last` the set of read states below $r$, which **may** be reached **last** when descending into $r$.

---

Berry-Sethi Approach: 1st step

$\text{empty}[r] = t$ if and only if $\epsilon \in [r]$

... for example:
Berry-Sethi Approach: 1st step

\[ \text{empty}[r] = t \text{ if and only if } \epsilon \in [r] \]

... for example:

![Diagram 1](image1)

Berry-Sethi Approach: 2nd step

\[ \text{empty}[r] = t \text{ if and only if } \epsilon \in [r] \]

... for example:

![Diagram 2](image2)

Implementation:

DFS (Post-order traversal)

for leaves \( r = [1, x] \) we find \( \text{empty}[r] = (x = \epsilon) \).

Otherwise:

\[
\begin{align*}
\text{empty}[r_1 | r_2] &= \text{empty}[r_1] \lor \text{empty}[r_2] \\
\text{empty}[r_1 \cdot r_2] &= \text{empty}[r_1] \land \text{empty}[r_2] \\
\text{empty}[r_1?] &= t \\
\text{empty}[r_1.?] &= t
\end{align*}
\]
Berry-Sethi Approach: 1st step

\[ \text{empty}[r] = t \quad \text{if and only if} \quad \varepsilon \in [r] \]

... for example:

The set of read states, that may be reached from \( r \) (i.e. while descending into \( r \)) via sequences of \( \epsilon \)-transitions:

\[ \text{first}[r] = \{ i \in r \mid (r, \epsilon, i \xrightarrow{\epsilon} x) \in \delta^*, x \neq \varepsilon \} \]

... for example:

---

Berry-Sethi Approach: 2nd step

Implementation:

DFS post-order traversal

For leaves \( r = \text{[leaf]} \), we find \( \text{empty}[r] = (x = \varepsilon) \).

Otherwise:

\[
\begin{align*}
\text{empty}[r_1 \upharpoonright r_2] &= \text{empty}[r_1] \lor \text{empty}[r_2] \\
\text{empty}[r_1 \cdot r_2] &= \text{empty}[r_1] \land \text{empty}[r_2] \\
\text{empty}[r_1 ?] &= t \\
\text{empty}[r_1 :] &= t
\end{align*}
\]

---

Berry-Sethi Approach: 2nd step

The set of read states, that may be reached from \( r \) (i.e. while descending into \( r \)) via sequences of \( \epsilon \)-transitions:

\[ \text{first}[r] = \{ i \in r \mid (r, \epsilon, i \xrightarrow{\epsilon} x) \in \delta^*, x \neq \varepsilon \} \]

... for example:
Berry-Sethi Approach: 2nd step

The may-set of first reached read state: The set of read states, that may be reached from $\epsilon$ (i.e. while descending into $r$) via sequences of $\epsilon$-transitions:

$\text{first}[r] = \{ i \in r \mid (\epsilon, \epsilon, i \rightarrow x) \in \delta^*, x \neq \epsilon \}$

... for example:

---

Berry-Sethi Approach: 3rd step

The may-set of next read states: The set of read states within the subtrees right of $r_{\epsilon}$ that may be reached next via sequences of $\epsilon$-transitions:

$\text{next}[r] = \{ i \mid (r_{\epsilon}, \epsilon, i \rightarrow x) \in \delta^*, x \neq \epsilon \}$

... for example:
Berry-Sethi Approach: 3rd step

The may-set of next read states: The set of read states within the subtrees right of \( r_e \), that may be reached next via sequences of \( \epsilon \)-transitions. 

\[ \text{next}[r] = \{ i \mid (r_e, \epsilon, \bullet r_i x) \in \delta^*, x \neq \epsilon \} \]

... for example:

---

Berry-Sethi Approach: 3rd step

The may-set of next read states: The set of read states within the subtrees right of \( r_e \), that may be reached next via sequences of \( \epsilon \)-transitions. 

\[ \text{next}[r] = \{ i \mid (r_e, \epsilon, \bullet r_i x) \in \delta^*, x \neq \epsilon \} \]

... for example:
Berry-Sethi Approach: 4th step

The may-set of last reached read states: The set of read states, which may be reached last during the traversal of $r$ connected to the root via $\epsilon$-transitions only:

$\text{last}[r] = \{ i \in r | (x \in \{\epsilon, r\} \in \delta^*, x \neq \epsilon) \}$

... for example:

Berry-Sethi Approach: (sophisticated version)

Construction (sophisticated version): Create an automaton based on the syntax tree's new attributes:

- **States**: $\{ \ast \epsilon \} \cup \{ i \ast | i \text{ a leaf} \}$
- **Start state**: $\ast\epsilon$
- **Final states**: $\text{last}[\epsilon]$ if $\emptyset \\{\epsilon\} = f$
  - $\{\ast \epsilon \} \cup \text{last}[\epsilon]$ otherwise
- **Transitions**: $\{\ast \epsilon, a, i \ast \}$ if $i \in \text{first}[\epsilon]$ and $i$ labeled with $a$.
  - $\{i \ast \ast, a \}$ if $i' \in \text{next}[i]$ and $i'$ labeled with $a$.

We call the resulting automaton $A_v$.

Remarks:
- This construction is known as Berry-Sethi or Glushkov construction.
- It is used for XML to define Content Models
- The result may not be, what we had in mind...
Chapter 4: Turning NFAs deterministic

Powerset Construction

... for example:

Powerset Construction

... for example:
For every non-deterministic automaton $A = (Q, \Sigma, \delta, I, F)$ we can compute a deterministic automaton $\mathcal{P}(A)$ with

$$\mathcal{L}(A) = \mathcal{L}(\mathcal{P}(A))$$

**Construction:**

- **States:** Powersets of $Q$;
- **Start state:** $I$;
- **Final states:** $\{Q' \subseteq Q \mid Q' \cap F \neq \emptyset\}$;
- **Transitions:** $\delta_P \left( Q', a \right) = \{ q \in Q \mid \exists p \in Q' : (p, a, q) \in \delta \}$

**Bummer!**
There are exponentially many powersets of $Q$

- Idea: Consider only contributing powersets. Starting with the set $Q_P = \{ I \}$ we only add further states by need ...
- i.e., whenever we can reach them from a state in $Q_P$
- Even though, the resulting automaton can become enormously huge
  ... which is (sort of) not happening in practice

Therefore, in tools like `grep` a regular expression’s DFA is never created!
Instead, only the sets, directly necessary for interpreting the input are generated while processing the input
Powerset Construction

... for example:

```
  a b a b
```

Remarks:

- For an input sequence of length $n$, maximally $O(n)$ sets are generated.
- Once a set/edge of the DFA is generated, they are stored within a hash-table.
- Before generating a new transition, we check this table for already existing edges with the desired label.