Lesson 6

- From RE to DFA, directly
- Minimization of DFA’s
- Exercises on lexical analysis
From Regular Expression to DFA Directly

• The “important states” of an NFA are those with a non-$\varepsilon$ outgoing transition,
  – if $\text{move}(\{s\}, a) \neq \emptyset$ for some $a$ then $s$ is an important state

• The subset construction algorithm uses only the important states when it determines $\varepsilon$-closure($\text{move}(T, a)$)
What are the “important states” in the NFA built from Regular Expression?

- $\varepsilon$
- $a$
- $r_1 \mid r_2$
- $r_1r_2$
- $r^*$
From Regular Expression to DFA Directly (Algorithm)

• The only accepting state (via the Thompson algorithm) is not important
• Augment the regular expression $r$ with a special end symbol $#$ to make accepting states important: the new expression is $r#$
• Construct a syntax tree for $r#$
• Attach a unique integer to each node not labeled by $\varepsilon$
From Regular Expression to DFA
Directly: Syntax Tree of  \((a|b)^*abb\#\)
From Regular Expression to DFA Directly: Annotating the Tree

- Traverse the tree to construct functions $\textit{nullable}$, $\textit{firstpos}$, $\textit{lastpos}$, and $\textit{followpos}$
- For a node $n$, let $L(n)$ be the language generated by the subtree with root $n$
- $\textit{nullable}(n)$: $L(n)$ contains the empty string $\varepsilon$
- $\textit{firstpos}(n)$: set of positions under $n$ that can match the first symbol of a string in $L(n)$
- $\textit{lastpos}(n)$: the set of positions under $n$ that can match the last symbol of a string in $L(n)$
- $\textit{followpos}(i)$: the set of positions that can follow position $i$ in any generated string
From Regular Expression to DFA Directly: Annotating the Tree

<table>
<thead>
<tr>
<th>Node $n$</th>
<th>$\text{nullable}(n)$</th>
<th>$\text{firstpos}(n)$</th>
<th>$\text{lastpos}(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leaf $\epsilon$</td>
<td>true</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>Leaf $i$</td>
<td>false</td>
<td>${i}$</td>
<td>${i}$</td>
</tr>
<tr>
<td>$</td>
<td>$ / \</td>
<td>$\text{nullable}(c_1)$ or $\text{nullable}(c_2)$</td>
<td>$\text{firstpos}(c_1) \cup \text{firstpos}(c_2)$</td>
</tr>
<tr>
<td>$\cdot$ / \</td>
<td>$\text{nullable}(c_1)$ and $\text{nullable}(c_2)$</td>
<td>$\text{if } \text{nullable}(c_1) \text{ then } \text{firstpos}(c_1) \cup \text{firstpos}(c_2) \text{ else } \text{firstpos}(c_1)$</td>
<td>$\text{if } \text{nullable}(c_2) \text{ then } \text{lastpos}(c_1) \cup \text{lastpos}(c_2) \text{ else } \text{lastpos}(c_2)$</td>
</tr>
<tr>
<td>$*$ $</td>
<td>$ / \</td>
<td>true</td>
<td>$\text{firstpos}(c_1)$</td>
</tr>
<tr>
<td>\ $c_1$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
From Regular Expression to DFA
Annotating the Syntax Tree of \((a|b)^*abb#\)
From Regular Expression to DFA

*followpos* on the Syntax Tree of \((a|b)^*abb\#\)
From Regular Expression to DFA Directly: \textit{followpos}

\begin{verbatim}
for each node $n$ in the tree do
    if $n$ is a cat-node with left child $c_1$ and right child $c_2$ then
        for each $i$ in $\text{lastpos}(c_1)$ do
            $\text{followpos}(i) := \text{followpos}(i) \cup \text{firstpos}(c_2)$
        end do
    else if $n$ is a star-node
        for each $i$ in $\text{lastpos}(n)$ do
            $\text{followpos}(i) := \text{followpos}(i) \cup \text{firstpos}(n)$
        end do
    end if
end do
\end{verbatim}
From Regular Expression to DFA Directly: Example

<table>
<thead>
<tr>
<th>Node</th>
<th>followpos</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>{1, 2, 3}</td>
</tr>
<tr>
<td>b</td>
<td>{1, 2, 3}</td>
</tr>
<tr>
<td>a</td>
<td>{4}</td>
</tr>
<tr>
<td>b</td>
<td>{5}</td>
</tr>
<tr>
<td>b</td>
<td>{6}</td>
</tr>
<tr>
<td>#</td>
<td>-</td>
</tr>
</tbody>
</table>
From Regular Expression to DFA Directly: The Algorithm

\[ s_0 := \text{firstpos}(\text{root}) \text{ where } \text{root} \text{ is the root of the syntax tree for } (r)\# \]

\[ \text{Dstates} := \{s_0\} \text{ and is unmarked} \]

\[ \text{while there is an unmarked state } T \text{ in Dstates do} \]

\[ \text{mark } T \]

\[ \text{for each input symbol } a \in \Sigma \text{ do} \]

\[ \text{let } U \text{ be the union of } \text{followpos}(p) \text{ for all positions } p \text{ in } T \]

\[ \text{such that the symbol at position } p \text{ is } a \]

\[ \text{if } U \text{ is not empty and not in Dstates then} \]

\[ \text{add } U \text{ as an unmarked state to Dstates} \]

\[ \text{end if} \]

\[ \text{Dtran}[T, a] := U \]

\[ \text{end do} \]

\[ \text{end do} \]
Minimizing the Number of States of a DFA

• Given a DFA, let us show how to get a DFA which accepts the same regular language with a minimal number of states
Equivalent States: Example

Consider the accept states c and g. They are both sinks.
Q: Do we need both states?
Equivalent States: Example

A: No, they can be merged!

Q: Can any other states be merged because any subsequent string suffixes produce identical results?
Equivalent States: Example

A: Yes, b and f. Notice that if you're in b or f then:
1. if string ends, reject in both cases
2. if next character is 0, forever accept in both cases
3. if next character is 1, forever reject in both cases
So merge b with f.
Equivalent States: Definition

Intuitively two states are equivalent if all subsequent behavior from those states is the same.

**DEF:** Two states $q$ and $q'$ in a DFA $M = (Q, \Sigma, \delta, q_0, F)$ are **equivalent** (or **indistinguishable**) if for all strings $u \in \Sigma^*$, the states on which $u$ ends on when read from $q$ and $q'$ are both **accept**, or both **non-accept**.
Finishing the Example

Q: Any other ways to simplify the automaton?
Useless States

A: Get rid of \( d \).

Getting rid of unreachable *useless states* doesn't affect the accepted language.
Minimization Algorithm: Goals

**DEF:** An automaton is *irreducible* if
- it contains no useless states, and
- no two distinct states are equivalent.

The goal of the **Minimization Algorithm** is to create an irreducible automata from an arbitrary one, accepting the same language.

The minimization algorithm incrementally builds a **partition** of the states of the given DFA:
- It starts with a partition separating just accepting/non accepting states
- Next it splits an equivalence class if it contains two non equivalent states
Minimization Algorithm.
(Partition Refinement) Code

DFA minimize(DFA (Q, S, d, q₀, F ))
remove any state q unreachable from q₀
Partition P = {F, Q - F }
boolean Consistent = false
while ( Consistent == false ) Consistent = true
  for(every Set S ∈ P, char a ∈ S, Set T ∈ P )
    // collect states of T that reach S using a
    Set temp = {q ∈ T | d(q,a) ∈ S }
    if (temp != Ø && temp != T )
      Consistent = false
      P = (P -T )∪{temp,T-temp}
return defineMinimizor( (Q, S, d, q₀, F ), P )
Minimization Algorithm.  
(Partition Refinement) Code

DFA defineMinimizor (DFA \((Q, \Sigma, \delta, q_0, F)\), Partition \(P\))

Set \(Q' = P\)

State \(q'_0 = \) the set in \(P\) which contains \(q_0\)

\(F' = \{ S \in P \mid S \subseteq F \}\)

for (each \(S \in P, a \in \Sigma\))

\[
\text{define } \delta'(S,a) = \text{ the set } T \in P \quad \text{which contains the states } \delta'(S,a)
\]

return \((Q', \Sigma, \delta', q'_0, F')\)
Minimization Algorithm: Example

Show the result of applying the minimization algorithm to this DFA
Proof of Minimal Automaton

Previous algorithm guaranteed to produce an **irreducible** DFA. Why should that FA be the smallest possible FA for its accepted language?

Analogous question in calculus: Why should a local minimum be a global minimum? *Usually not the case!*
Proof of Minimal Automaton

THM (Myhill-Nerode): *The minimization algorithm produces the smallest possible automaton for its accepted language.*

*Proof.* Show that any irreducible automaton is the smallest for its accepted language $L$:

We say that two strings $u,v \in \Sigma^*$ are *indistinguishable* if for all strings $x$, $ux \in L \iff vx \in L$.

Notice that if $u$ and $v$ are *distinguishable*, their paths from the start state must have different endpoints.
Proof of Minimal Automaton

Consequently, the number of states in any DFA for $L$ must be as great as the number of mutually distinguishable strings for $L$.

But an irreducible DFA has the property that every state gives rise to another mutually distinguishable string!

Therefore, any other DFA must have at least as many states as the irreducible DFA.

Let’s see how the proof works on a previous example:
Proof of Minimal Automaton: Example

The “spanning tree of strings” \(\{\varepsilon, 0, 01, 00\}\) is a mutually distinguishable set (otherwise redundancy would occur and hence DFA would be reducible). Any other DFA for \(L\) has \(\geq 4\) states.

![Diagram](image)
Exercises on Lexical Analysis

3.1.1 Divide the following C++ program into appropriate lexemes:

```c++
float limitedSquare(x){float x;
    /* returns x-squared, but never more than 100 */
    return (x <= -10.0 || x >= 10.0) ? 100 : x*x;
}
```

Which lexemes should get associated lexical values? What should those values be?
From RE to Automata and backwards

• We have seen:
  – RE → NFA
  – NFA → DFA [and obviously DFA → NFA]
  – RE → DFA, directly
  – DFA → minimal DFA

• What about NFA, DFA → RE? More difficult. Three approaches (not presented):
  • Dynamic Programming [Scott Section 2.4 on CD][Hopcroft, Motwani, Ullman, Section 3.2.1]
  • Incremental state elimination [HMU, Section 3.2.2]
  • RE as fixpoint solution of system of language equations
    [uses right-linear grammars for Regular Languages]
Exercises on Regular Expressions

3.3.2 Describe the languages denoted by the following regular expressions:
   b) \((\epsilon|a)b^*)^*\)
   c) \((a|b)^*a(a|b)(a|b)\)

3.3.5 Write regular definitions for the following languages:
   b) All strings of lowercase letters in which the letters are in ascending lexicographic order.
   c) Comments, consisting of a string surrounded by /* and */ , without an intervening */ , unless it is inside double-quotes (”)
   i) All strings of a's and b's that do not contain the subsequence abb.
Exercises with Lex or Flex

• 3.5.2 Write a Lex program that copies a file, replacing each non-empty sequence of white spaces by a single blank.

• 3.5.3 Write a Lex program that copies a C program, replacing each instance of the keyword float by double.
Exercises on Finite Automata

• 3.6.2 Design finite automata for the following languages (providing both the transition graph and the transition table):
  a) All strings of lowercase letters that contain the five vowels in order.
  d) All strings of digits with no repeated digits. Hint: Try this problem first with a few digits, such as \{0, 1, 2\}.
  f) All strings of a's and b's with an even number of a's and an odd number of b's.
Exercises: from RE to DFA

3.7.3 Convert the following regular expressions to deterministic finite automata, using the [McNaughton-Yamada-]Thompson algorithm (3.23) and the subset construction algorithm (3.20):

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

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

c) \(((\epsilon\|a)\|b^*)^*\)

d) \((a\|b)^*abb(a\|b)^*\)
Exercises: Minimizing DFA

• 3.9.3 Show that the RE
  a) \((a | b)^*\)
  b) \((a^* | b^*)^*\)
  c) \(((\varepsilon | a) | b^*)^*\)

are equivalent by showing that their minimum state DFA’s are isomorphic.