THIS COMPUTER SCIENTIST IS BELIEVED TO BE THE FIRST PERSON TO RECOGNISE THAT COMPUTERS HAD APPLICATIONS BEYOND CALCULATION AND IS OFTEN CREDITED TO HAVE PUBLISHED THE FIRST ALGORITHM TO BE CARRIED OUT ON A COMPUTER
Nondeterministic Finite Automata

CS 536
Previous Lecture

Scanner: converts a sequence of characters to a sequence of tokens

Scanner and parser relationship

Scanner implemented using FSMs

FSM: DFA or NFA
This Lecture

NFAs from a formal perspective
Theorem: NFAs and DFAs are equivalent
Regular languages and Regular expressions
NFAs, formally

\[ M \equiv (Q, \Sigma, \delta, q, F) \]

- finite set of states
- the alphabet (characters)
- transition function \( \delta : Q \times \Sigma \to 2^Q \)
- final states \( F \subseteq Q \)
- start state \( q \in Q \)

Transition function table:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>{s1}</td>
<td>{s1, s2}</td>
</tr>
<tr>
<td>s2</td>
<td>{}</td>
<td>{}</td>
</tr>
</tbody>
</table>
NFA

To check if string is in $L(M)$ of NFA $M$, simulate set of choices it could make

At least one sequence of transitions that:

- Consumes all input (without getting stuck)
- Ends in one of the final states
NFA and DFA are Equivalent

Two automata $M$ and $M'$ are equivalent iff $L(M) = L(M')$.

Lemmas to be proven

**Lemma 1**: Given a DFA $M$, one can construct an NFA $M'$ that recognizes the same language as $M$, i.e., $L(M') = L(M)$.

**Lemma 2**: Given an NFA $M$, one can construct a DFA $M'$ that recognizes the same language as $M$, i.e., $L(M') = L(M)$. 
Lemma 2: Given an NFA $M$, one can construct a DFA $M'$ that recognizes the same language as $M$, i.e., $L(M') = L(M)$

Idea: we can only be in finitely many subsets of states at any one time

$2^{|Q|}$ possible combinations of states

Why?
Why $2^{|Q|}$ states?

Build DFA that tracks set of states the NFA is in!

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
**Defn:** let $\text{succ}(s,c)$ be the set of choices the NFA could make in state $s$ with character $c$

\[
\begin{align*}
\text{succ}(A,x) &= \{A, B\} \\
\text{succ}(A,y) &= \{A\} \\
\text{succ}(B,x) &= \{C\} \\
\text{succ}(B,y) &= \{C\} \\
\text{succ}(C,x) &= \{D\} \\
\text{succ}(C,y) &= \{D\}
\end{align*}
\]
Build new DFA $M'$ where $Q' = 2^Q$

To build DFA: Add an edge from state $S$ on character $c$ to state $S'$ if $S'$ represents the union of states that all states in $S$ could possibly transition to on input $c$. 

- $\text{succ}(A,x) = \{A,B\}$
- $\text{succ}(A,y) = \{A\}$
- $\text{succ}(B,x) = \{C\}$
- $\text{succ}(B,y) = \{C\}$
- $\text{succ}(C,x) = \{D\}$
- $\text{succ}(C,y) = \{D\}$
**ε-transitions**

**Eg:** $x^n$, where $n$ is even or divisible by 3

Useful for taking union of two FSMs

In example, left side accepts even $n$; right side accepts $n$ divisible by 3
Eliminating ε-transitions

We want to construct ε-free FSM M’ that is equivalent to M.

**Definition:**
eclose(s) = set of all states reachable from s in zero or more epsilon transitions

**M’ components**
s is an accepting state of M’ iff eclose(s) contains an accepting state

s —c—> t is a transition in M’ iff q —c—> t for some q in eclose(s)
Eliminating ε-transitions

We want to construct ε-free NFA M’ that is equivalent to M

**Definition: Epsilon Closure**

eclose(s) = set of all states reachable from s using zero or more epsilon transitions

<table>
<thead>
<tr>
<th>State</th>
<th>ε-close</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>{A, B, D}</td>
</tr>
<tr>
<td>B</td>
<td>{B}</td>
</tr>
<tr>
<td>C</td>
<td>{C}</td>
</tr>
<tr>
<td>D</td>
<td>{D}</td>
</tr>
<tr>
<td>E</td>
<td>{E}</td>
</tr>
<tr>
<td>F</td>
<td>{F}</td>
</tr>
</tbody>
</table>
**Def**: \( \text{eclose}(s) = \text{set of all states reachable from } s \text{ in zero or more epsilon transitions} \)

\( s \) is an accepting state of \( M' \) iff \( \text{eclose}(s) \) contains an accepting state

\( s \xrightarrow{-c-} t \) is a transition in \( M' \) iff

\( q \xrightarrow{-c-} t \) for some \( q \) in \( \text{eclose}(s) \)
Recap

NFAs and DFAs are equally powerful

- any language definable as an NFA is definable as a DFA
- \(\varepsilon\)-transitions do not add expressiveness to NFAs

We showed a simple algorithm to remove epsilons
Regular Languages and Regular Expressions
Regular Language

Any language recognized by an FSM is a regular language

Examples:

- Single-line comments beginning with //</p></li>
- Integer literals
- \{\varepsilon, ab, abab, ababab, abababab, \ldots \}
- C/C++ identifiers
Regular expressions

Pattern describing a language

**operands:** single characters, epsilon

**operators:** from low to high precedence
- alternation “or”: $a \mid b$
- catenation: $a.b$, $ab$, $a^3$ (which is aaa)
- iteration: $a^*$ (0 or more a’s) aka Kleene star
Why do we need them?

Each token in a programming language can be defined by a regular language

Scanner-generator input: one regular expression for each token to be recognized by scanner

Regular expressions are inputs to a scanner generator
Regexp, cont’d

Conventions:

- $a^+$ is $aa^*$
- letter is $a|b|c|d|...|y|z|A|B|...|Z$
- digit is $0|1|2|...|9$
- not(x) all characters except $x$
- . is any character
- parentheses for grouping, e.g., $(ab)^*$
  - $\varepsilon, ab, abab, ababab$
Regexp, example

Hex strings
- start with 0x or 0X
- followed by one or more hexadecimal digits
- optionally end with l or L

0(x|X)hexdigit+(L|l|ε)
where hexdigit = digit|a|b|c|d|e|f|A|…|F
Regexp, example

Single-line comments in Java/C/C++
   // this is a comment
   //(not(‘\n’))*(’\n’|epsilon)
Regexp, example

C/C++ identifiers: sequence of letters/digits/underscores; cannot begin with a digit; cannot end with an underscore

Example: a, _bbb7, cs_536

Regular expression

letter | (letter|_)(letter|digit|_)* (letter|digit)
Recap

Regular Languages
- Languages recognized/defined by FSMs

Regular Expressions
- Single-pattern representations of regular languages
- Used for defining tokens in a scanner generator
Creating a Scanner

Last lecture: DFA to code

This lecture: NFA to DFA

Next lecture: Regexp to NFA

This lecture: token to Regexp

Scanner Generator

Scanner