NFA vs. DFA

For every NFA, there is a DFA that accepts the same set of strings,

- NFA may have transitions labeled by \( \varepsilon \).
  (Spontaneous transitions)
- All transition labels in a DFA belong to \( \Sigma \).
- For some string \( x \), there may be many accepting paths in an NFA.
- For all strings \( x \), there is one unique accepting path in a DFA.
- Usually, an input string can be recognized faster with a DFA.
- NFAs are typically smaller than the corresponding DFAs.

Regular Expressions to NFA

Thompson’s Construction: For every regular expression \( r \), derive an NFA \( N(r) \) with unique start and final states.

\[
\begin{align*}
\epsilon & \quad \cdots \\
\alpha & \quad \cdots \\
(r_1 | r_2) & \quad \cdots \\
\end{align*}
\]

Recognition with an NFA

Is \( abab \in L((a | b)^*a(a | b)) \)?

\[
\begin{array}{c}
\begin{array}{c}
\text{Input:} \\
\text{Path 1:} \\
\text{Path 2:} \\
\text{Path 3:} \\
\end{array}
\end{array}
\begin{array}{c}
a \ b \ a \ b \\
1 \ 1 \ 1 \ 1 \ \\
1 \ 1 \ 1 \ 2 \ \checkmark \\
1 \ 2 \ 3 \ \xmark \ \xmark \\
\end{array}
\]

Accept

Recognition with a DFA

Is \( abab \in L((a | b)^*a(a | b)) \)?

\[
\begin{array}{c}
\begin{array}{c}
\text{Input:} \\
\text{Path:} \\
\end{array}
\end{array}
\begin{array}{c}
a \ b \ a \ b \\
1 \ a \ a \ b \\
1 \ 2 \ 4 \ 2 \ \checkmark \\
\end{array}
\]

Accept
Recognition with an NFA

Is $aabab \in \mathcal{L}((a \mid b)^*a(a \mid b))$?

Input:
Path 1: \[ a \mid b \mid a \mid b \]
Path 2: \[ a \mid b \mid 1 \mid 1 \mid 2 \mid 3 \] Accept
Path 3: \[ 1 \mid 2 \mid 3 \] 

All Paths: \[ \{1\} \mid \{1,2\} \mid \{1,3\} \mid \{1,2\} \mid \{1,3\} \] Accept

Regular Expressions to NFA (cont'd.)

$R_1 \cdot R_2$

Example

$(a \mid b)^*a(a \mid b)$:
Converting NFA to DFA (contd.).

Each state in DFA corresponds to a set of states in NFA.
Start state of DFA = \(\epsilon\)-closure(start state of NFA).
From a state \(s\) in DFA that corresponds to a set of states \(S\) in NFA:

- add a transition labeled \(\alpha\) to state \(s'\) that corresponds to a non-empty \(S'\) in NFA,
- such that \(S' = \text{goto}(S, \alpha)\),

\(s\) is a state in DFA such that the corresponding set of states \(S\) in NFA contains a final state of NFA,
\(\Leftrightarrow s\) is a final state of DFA

NFA \(\rightarrow\) DFA: An Example

\(\epsilon\)-closure(\{1\}) = \{1\}
goto(\{1\}, a) = \{1, 2\}
goto(\{1\}, b) = \{1\}
goto(\{1, 2\}, a) = \{1, 2, 3\}
goto(\{1, 2\}, b) = \{1, 3\}
goto(\{1, 2, 3\}, a) = \{1, 2, 3\}

Recognition with an NFA (contd.)

Is \(aabb \in L((a | b)^*a(a | b))\)?

Input: \ a \ a \ a \ b \\
Path 1: \ 1 \ 1 \ 1 \ 1 \ 1 \\
Path 2: \ 1 \ 1 \ 2 \ 3 \ \downarrow \\
Path 3: \ 1 \ 2 \ 3 \ \downarrow \ \downarrow \\
All Paths: \ \{1\} \ \{1, 2\} \ \{1, 2, 3\} \ \{1, 3\} \ \{1\} \ REJECT

Converting NFA to DFA

Subset construction
Given a set \(S\) of NFA states,
- compute \(S_0 = \epsilon\)-closure\((S)\): \(S_0\) is the set of all NFA states reachable by zero or more \(\epsilon\)-transitions from \(S\),
- compute \(S_\alpha = \text{goto}(S, \alpha)\):
  - \(S'\) is the set of all NFA states reachable from \(S\) by taking a transition labeled \(\alpha\),
  - \(S_\alpha = \epsilon\)-closure\((S')\),
NFA vs. DFA

\( R = \text{Size of Regular Expression} \)
\( N = \text{Length of Input String} \)

<table>
<thead>
<tr>
<th></th>
<th>NFA</th>
<th>DFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of Automaton</td>
<td>( O(R) )</td>
<td>( O(2^R) )</td>
</tr>
<tr>
<td>Recognition time per input string</td>
<td>( O(N \times R) )</td>
<td>( O(N) )</td>
</tr>
</tbody>
</table>

Lexical Analysis

- Regular Expressions and Definitions are used to specify the set of strings (lexemes) corresponding to a token.
- An automaton (DFA/NFA) is built from the above specifications.
- Each final state is associated with an action: emit the corresponding token.

NFA \(\rightarrow\) DFA: An Example (contd.)

- \( \text{closure}(\{1\}) = \{1\} \)
- \( \text{goto}(\{1\}, a) = \{1, 2\} \)
- \( \text{goto}(\{1\}, b) = \{1\} \)
- \( \text{goto}(\{1, 2\}, a) = \{1, 2, 3\} \)
- \( \text{goto}(\{1, 2\}, b) = \{1, 3\} \)
- \( \text{goto}(\{1, 2, 3\}, a) = \{1, 2, 3\} \)
- \( \text{goto}(\{1, 2, 3\}, b) = \{1\} \)
- \( \text{goto}(\{1, 3\}, a) = \{1, 2\} \)
- \( \text{goto}(\{1, 3\}, b) = \{1\} \)

\[\begin{array}{cccc}
\text{state} & a & b & \text{action} \\
\{1\} & a & b & \{1\} \\
\{1, 2\} & a & b & \{1, 2, 3\} \\
\{1, 3\} & a & b & \{1, 2\} \\
\{1, 2, 3\} & a & b & \{1, 2, 3\}
\end{array}\]
Lex Specifications

% (  
   C header statements for inclusion
%
)
Regular Definitions e.g.:
digit [0-9]
%
Token Specifications e.g.:
digit* { return(INTEGER_CONSTANT); }
%
Support functions in C

Regular Expressions in Lex

Adds "syntactic sugar" to regular expressions:
- Range: [0-7]: Integers from 0 through 7 (inclusive)
  [a-zA-Z]: Letters a thru n, x thru z and A thru Q.
- Exception: [^/]: Any character other than /.
- Definition: {digit}: Use the previously specified regular
definition digit.
- Special characters: Connectives of regular expression,
  convenience features,
e.g.: | * ^

Specifying Lexical Analysis

Consider a recognizer for integers (sequence of digits) and floats
(sequence of digits separated by a decimal point).

\[
[0-9]+ \quad \{ \text{emit(INTEGER_CONSTANT);} \}
\]

\[
[0-9]+ \cdot[0-9]+ \quad \{ \text{emit(FLOAT_CONSTANT);} \}
\]

Lex

Tool for building lexical analyzers,
Input: lexical specifications (.l file)
Output: C function (yylex) that returns a token on each
invocation,

% 
[0-9]+ \quad \{ \text{return(INTEGER_CONSTANT);} \}
%

\[
[0-9]+ \cdot[0-9]+ \quad \{ \text{return(FLOAT_CONSTANT);} \}
\]

Tokens are simply integers (#defines).
A Complete Example

```c
#include <stdio.h>
#include "tokens.h"
%
digit [0-9]
handigit [0-9a-f]
%
"+" { return(PLUS); }
"-" { return(MINUS); }
{digit}+ { return(INTEGER_CONSTANT); }
{digit}"{digit}+ { return(REAL_CONSTANT); }
. { return(SYNTAX_ERROR); }
%
```

Special Characters in Lex

<table>
<thead>
<tr>
<th>*</th>
<th>+</th>
<th>?</th>
<th>(</th>
<th>)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Same as in regular expressions

Enclose ranges and exceptions

Enclose "names" of regular definitions

Used to negate a specified range (in Exception)

Match any single character except newline

Escape the next character

Newline and Tab

For literal matching, enclose special characters in double quotes ("")

*e.g.*: "*"

Or use \ to escape, *e.g.*: \\

Examples

<table>
<thead>
<tr>
<th>for</th>
<th>Sequence of f, o, r</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;[&quot;</td>
<td>C-style OR operator (two vertical bars)</td>
</tr>
<tr>
<td>.*</td>
<td>Sequence of non-newline characters</td>
</tr>
<tr>
<td>[&quot;*/&quot;]+</td>
<td>Sequence of characters except * and /</td>
</tr>
<tr>
<td>&quot;[&quot;&quot;]&quot;</td>
<td>Sequence of non-quote characters beginning and ending with a quote</td>
</tr>
<tr>
<td>({letter}&quot;&quot;)({letter}{digit}&quot;&quot;)*</td>
<td>C-style identifiers</td>
</tr>
</tbody>
</table>