Lexical Analysis
Part III

Chapter 3: Finite Automata

Slides adapted from:
© Robert van Engelen, Florida State University
Alex Aiken, Stanford University
Design of a Lexical Analyzer Generator

- Translate regular expressions to NFA
- Translate NFA to an efficient DFA

Diagram:

```
regular expressions -> NFA -> DFA

Simulate NFA to recognize tokens
Simulate DFA to recognize tokens
```

Optional
Nondeterministic Finite Automata

- An **nondeterministic finite automaton** (NFA) is a 5-tuple \((S, \Sigma, \delta, s_0, F)\) where

  - \(S\) is a finite set of *states*
  - \(\Sigma\) is a finite set of symbols, the *alphabet*
  - \(\delta\) is a *mapping* from \(S \times (\Sigma \cup \{\epsilon\})\) to a subset of \(S\)
  - \(s_0 \in S\) is the *start state*
  - \(F \subseteq S\) is the set of *accepting* (or *final*) *states*
• An NFA can be diagrammatically represented by a labeled directed graph called a transition graph

Transition Graph

\[ S = \{0,1,2,3\} \]
\[ \Sigma = \{a,b\} \]
\[ s_0 = 0 \]
\[ F = \{3\} \]
The mapping $\delta$ of an NFA can be represented in a transition table.

<table>
<thead>
<tr>
<th>State</th>
<th>Input $a$</th>
<th>Input $b$</th>
<th>Input $\epsilon$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>${0, 1}$</td>
<td>${0}$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>${2}$</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>${3}$</td>
<td></td>
</tr>
</tbody>
</table>

- $\delta(0, a) = \{0, 1\}$
- $\delta(0, b) = \{0\}$
- $\delta(1, b) = \{2\}$
- $\delta(2, b) = \{3\}$
The Language Defined by an NFA

• An NFA accepts an input string $x$ if and only if there is some path with edges labeled with symbols from $x$ in sequence from the start state to some accepting state

• A state transition from one state to another on the path is called a move

• The language defined by an NFA is the set of input strings it accepts
Example NFA

A NFA that accepts $L( \text{aa}^* | \text{bb}^* )$
Deterministic Finite Automata

- A deterministic finite automaton (DFA) is a special case of an NFA
  - No state has an \(\varepsilon\)-transition
  - For each state \(s\) and input symbol \(a\) there is at most one edge labeled \(a\) leaving \(s\)
- Each entry in the transition table is a single state or is undefined
  - At most one path exists to accept a string
  - Simulation algorithm is simple
Example DFA

A DFA that accepts $L( (a|b)^*abb )$
Exercise

• Select the regular language that denotes the same language as this finite automaton

\[- 1^* \mid (01)^* \mid (001)^* \mid (000*1)^*\]
\[- (0 \mid 1)^*\]
\[- (1^* \mid 0)(1 \mid 0)\]
\[- (0 \mid 1)^*00\]
Exercise

• Choose the NFA that accepts the following regular expression: $1^* \mid 0$
Simulating a DFA

\[ s = s_0; \]
\[ c = \text{nextChar}(); \]
\[ \text{while ( c \neq \text{eof} ) } \{ \]
\[ \quad s = \text{move}(s, c); \]
\[ \quad c = \text{nextChar}(); \]
\[ \} \]
\[ \text{if ( s in F ) return “yes”; } \]
\[ \text{else return “no”; } \]
Design of a Lexical Analyzer Generator: RE to NFA to DFA

Lex specification with regular expressions

\[ p_1 \{ action_1 \} \]
\[ p_2 \{ action_2 \} \]
\[ \ldots \]
\[ p_n \{ action_n \} \]

NFA

DFA

Subset construction
From Regular Expression to NFA

\[ \varepsilon \]

\[ a \]

\[ r_1 \mid r_2 \]

\[ r_1r_2 \]

\[ r^* \]
Example:
Construct the NFA for $a \ (b|c)^*$

First: NFAs for $a, b, c$

Second: NFA for $blc$

Third: NFA for $(blc)^*$
Fourth: NFA for $a(b|c)^*$

Example:
Construct the NFA for $a(b|c)^*$

Of course, a human would design a simpler one... But, we can automate production of the complex one...
Combining the NFAs of a Set of Regular Expressions

\[
\begin{align*}
a & \quad \{ \text{action}_1 \} \\
abb & \quad \{ \text{action}_2 \} \\
a^*b^+ & \quad \{ \text{action}_3 \}
\end{align*}
\]
Simulating the Combined NFA

Example 1

Must find the longest match:
Continue until no further moves are possible
When last state is accepting: execute action
Simulating the Combined NFA
Example 2

When two or more accepting states are reached, the first action given in the Lex specification is executed.
Errors

• What if no rule matches?
• Create a new state in the automaton corresponding to the regular expression “all strings not in the lexical specification”
• Put the regular expression last in priority
Auxiliary Functions:

\( \varepsilon\text{-closure}() \) and \( \text{move}() \)

- Used in several constructions later:

\[
\varepsilon\text{-closure}(s) = \{s\} \cup \{t \mid s \xrightarrow{\varepsilon} \cdots \xrightarrow{\varepsilon} t\} \\
\varepsilon\text{-closure}(T) = \bigcup_{s \in T} \varepsilon\text{-closure}(s) \\
\text{move}(T, a) = \{t \mid s \xrightarrow{a} t \text{ and } s \in T\}
\]
Examples for \(\varepsilon\)-closure() and move()

\[\varepsilon\text{-closure}\{0\} = \{0,1,3,7\}\]
\[\text{move}\{0,1,3,7\}, а = \{2,4,7\}\]
\[\varepsilon\text{-closure}\{2,4,7\} = \{2,4,7\}\]
\[\text{move}\{2,4,7\}, а = \{7\}\]
\[\varepsilon\text{-closure}\{7\} = \{7\}\]
\[\text{move}\{7\}, b = \{8\}\]
\[\varepsilon\text{-closure}\{8\} = \{8\}\]
\[\text{move}\{8\}, а = \emptyset\]
Simulating an NFA using $\varepsilon$-closure() and move()

$S = \varepsilon$-closure($s_0$);
$c = \text{nextChar}()$;
while ( $c \neq \text{eof}$ ) {
    $S = \varepsilon$-closure(move($S$, $c$));
    $c = \text{nextChar}()$;
}
if ( $S \cap F \neq \emptyset$ ) return “yes”;
else return “no”;
Simulating an NFA: Additional Data Structure

• Two stacks:
  – $oldStates$ holds current set of states
  – $newStates$ holds next set of states

• Boolean array $alreadyOn$, indexed by NFA states, indicates which states are in $newStates$

• Two-dimensional array $move[s, a]$ representing the transition table
Simulating an NFA: Auxiliary Function

\[addState(s) \{\]
  \[\text{push } s \text{ onto } newStates;\]
  \[alreadyOn[s] = \text{TRUE};\]
  \[\text{for ( } t \text{ on } move[s, \epsilon] \text{ )} \]
    \[\text{if ( } ! alreadyOn[t] \text{ )} \]
    \[addState(t);\]
\[\}\]
Simulating an NFA: Auxiliary Code

```java
for ( s on oldStates ) {
    for ( t on move[s, c] )
        if ( ! alreadyOn[t] )
            addState(t);
    pop s from oldStates;
}
```
Simulating an NFA: 
Auxiliary Code

```c
for ( s on newStates ) {
    pop s from newStates;
    push s onto oldStates;
    alreadyOn[s] = FALSE;
}
```
Simulating an NFA: Stream Version

\[ S = \varepsilon\text{-closure}\{s_0\}; \]
\[ S_{prev} = \emptyset; \]
\[ c = \text{nextChar}(); \]
\[ \text{while } (S \neq \emptyset) \{ \]
  \[ \text{if } (S \cap F \neq \emptyset) \]
  \[ S_{prev} = S; \]
  \[ S = \varepsilon\text{-closure}\text{move}(S, c); \]
  \[ c = \text{nextChar}(); \}
\[ \text{if } (S_{prev} \neq \emptyset) \{ \]
  \[ \text{execute highest priority action in } S_{prev}; \text{return “yes”}; \}
\[ \text{else return “error”;} \]
The Subset Construction Algorithm

- Off-line version of the algorithm for simulation of NFAs on a full word
- The algorithm produces:
  - $Dstates$, the set of states of the new DFA consisting of sets of states of the NFA
  - $Dtran$, the transition table of the new DFA
The Subset Construction Algorithm

add $\varepsilon$-closure($s_0$) as an unmarked state to $Dstates$

while (there is an unmarked state $T$ in $Dstates$) {
  mark $T$;
  for each input symbol $a \in \Sigma$ {
    $U = \varepsilon$-closure(move($T$, $a$));
    if ($U$ is not in $Dstates$)
      add $U$ as an unmarked state to $Dstates$
    $Dtran[T, a] = U$;
  }
}
Subset Construction Example 1

Dstates
A = \{0,1,2,4,7\}
B = \{1,2,3,4,6,7,8\}
C = \{1,2,4,5,6,7\}
D = \{1,2,4,5,6,7,9\}
E = \{1,2,4,5,6,7,10\}
Subset Construction Example 2

\[D\text{states}\]
A = \{0,1,3,7\}
B = \{2,4,7\}
C = \{8\}
D = \{7\}
E = \{5,8\}
F = \{6,8\}
Exercise

- Choose the DFA that represents the same language as the given NFA
Recap

Decision procedure for string $s$ and regular expression $R$

1. Generate NFA from $R$
2. Either:
   – Convert NFA to DFA
   – Run DFA simulation algorithm on $s$
3. Or:
   – Run NFA simulation algorithm on $s$
Time-Space Tradeoffs

- \( r \) regular expression, \( x \) input string

<table>
<thead>
<tr>
<th>Automaton</th>
<th>Space (worst case)</th>
<th>Time (worst case)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFA</td>
<td>( O(</td>
<td>r</td>
</tr>
<tr>
<td>DFA</td>
<td>( O(2^{</td>
<td>r</td>
</tr>
</tbody>
</table>
From Regular Expression to DFA Directly

• The “important states” of an NFA are those without an ε-transition, that is if \( move(\{s\}, a) \neq \emptyset \) for some \( a \) then \( s \) is an important state

• The subset construction algorithm uses only the important states when it determines \( \varepsilon\text{-closure}(move(T, a)) \)
From Regular Expression to DFA Directly (Algorithm)

- 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 $r#$
- Traverse the tree to construct functions $\text{nullable}$, $\text{firstpos}$, $\text{lastpos}$, and $\text{followpos}$
From Regular Expression to DFA Directly: Syntax Tree of \((a|b)^*abb#\)
From Regular Expression to DFA Directly: Annotating the Tree

- **nullable(n):** the subtree at node $n$ generates languages including the empty string
- **firstpos(n):** set of positions that can match the first symbol of a string generated by the subtree at node $n$
- **lastpos(n):** the set of positions that can match the last symbol of a string generated by the subtree at node $n$
- **followpos(i):** the set of positions that can follow position $i$ in the tree
### 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>$\mid$ $/$ $\backslash$ $c_1$ $c_2$</td>
<td>$\text{nullable}(c_1)$ or $\text{nullable}(c_2)$</td>
<td>$\text{firstpos}(c_1) \cup \text{firstpos}(c_2)$</td>
<td>$\text{lastpos}(c_1) \cup \text{lastpos}(c_2)$</td>
</tr>
<tr>
<td>$\cdot$ $/$ $\backslash$ $c_1$ $c_2$</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>$*$ $\mid$ $c_1$</td>
<td>true</td>
<td>$\text{firstpos}(c_1)$</td>
<td>$\text{lastpos}(c_1)$</td>
</tr>
</tbody>
</table>
From Regular Expression to DFA Directly: 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 lastpos(\( c_1 \)) do
            followpos(\( i \)) := followpos(\( i \)) \cup firstpos(\( c_2 \))
        end do
    else if \( n \) is a star-node
        for each \( i \) in lastpos(\( n \)) do
            followpos(\( i \)) := followpos(\( i \)) \cup firstpos(\( n \))
        end do
    end if
end do
\end{verbatim}
From Regular Expression to DFA Directly: Algorithm

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

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

\[ \textbf{while} \text{ there is an unmarked state } T \text{ in } D\text{states} \text{ do} \]

\[ \text{mark } T \]

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

\[ \text{let } U \text{ be the set of positions that are in } \text{followpos}(p) \]

\[ \text{for some position } p \text{ in } T, \]

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

\[ \textbf{if} \text{ } U \text{ is not empty and not in } D\text{states} \text{ then} \]

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

\[ \text{end if} \]

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

\[ \text{end do} \]

\[ \text{end do} \]
From Regular Expression to DFA Directly: Example

<table>
<thead>
<tr>
<th>Node</th>
<th>followpos</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{1, 2, 3}</td>
</tr>
<tr>
<td>2</td>
<td>{1, 2, 3}</td>
</tr>
<tr>
<td>3</td>
<td>{4}</td>
</tr>
<tr>
<td>4</td>
<td>{5}</td>
</tr>
<tr>
<td>5</td>
<td>{6}</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
</tr>
</tbody>
</table>
Implementing Transition Function

• Two-dimensional table indexed by current state and input character
• Several rows might be equal
• Compress table by using an array indexed by current state, providing a pointer to an array indexed by input character
Implementing Transition Function

• Alternatively, use adjacency matrix

• For each state, record a list of transitions in the form of input character-state pairs

• List ended by a default state for any input character not on the list
Implementing Transition Function

• Four array solution