Lexical Analysis
Part II
Chapter 3

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

regular expressions → NFA → DFA

Simulate NFA to recognize tokens
Optional
Simulate DFA to recognize tokens
Nondeterministic Finite Automata

- An 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 set of states
  \(s_0 \in S\) is the start state
  \(F \subseteq S\) is the set of accepting (or final) states

Transition Graph

- An NFA can be diagrammatically represented by a labeled directed graph called a transition graph

\[ S = \{0,1,2,3\} \]
\[ \Sigma = \{a,b\} \]
\[ s_0 = 0 \]
\[ F = \{3\} \]
Transition Table

- 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>${2}$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>${3}$</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 in the transition graph
- 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, such as $L( (a | b)^* abb )$ for the example NFA
Example NFA

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

Deterministic Finite Automata

- A deterministic finite automaton 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
  - 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

- $1^* + (01)^* + (001)^* + (000^*1)^*$
- $(0 + 1)^*$
- $(1^* + 0)(1 + 0)$
- $(0 + 1)^*00$
Exercise

• Choose the NFA that accepts the following regular expression: $1^* + 0$

Simulating a DFA

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

Lex specification with regular expressions

\[ p_1 \{ \text{action}_1 \} \]
\[ p_2 \{ \text{action}_2 \} \]
\[ \ldots \]
\[ p_n \{ \text{action}_n \} \]

NFA

\[ \text{start} \]
\[ \varepsilon \]
\[ \text{N}(p_1) \]
\[ \text{action}_1 \]
\[ \varepsilon \]
\[ \text{N}(p_2) \]
\[ \text{action}_2 \]
\[ \ldots \]
\[ \varepsilon \]
\[ \text{N}(p_n) \]
\[ \text{action}_n \]

Subset construction

DFA

From Regular Expression to NFA

\[ \varepsilon \]
\[ \text{start} \]
\[ \text{start} \]
\[ \varepsilon \]
\[ \varepsilon \]

\[ a \]
\[ \text{start} \]
\[ \text{start} \]
\[ \text{start} \]

\[ r_1 | r_2 \]
\[ \text{start} \]
\[ \varepsilon \]
\[ \varepsilon \]
\[ \text{N}(r_1) \]
\[ \text{N}(r_2) \]

\[ r_1 r_2 \]
\[ \text{start} \]
\[ \varepsilon \]
\[ \varepsilon \]
\[ \text{N}(r_1) \]
\[ \text{N}(r_2) \]

\[ r^* \]
\[ \text{start} \]
\[ \varepsilon \]
\[ \varepsilon \]
\[ \text{N}(r) \]
Example: Construct the NFA for $a \ (b | c)^*$

First: NFAs for $a$, $b$, $c$

Second: NFA for $b | c$

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

Fourth: 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

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

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
Conversion of an NFA into a DFA

- The subset construction algorithm converts an NFA into a DFA using:

  \[ \varepsilon\text{-closure}(s) = \{s\} \cup \{t \mid s \to^* \varepsilon \cdots \to^* \varepsilon t\} \]

  \[ \varepsilon\text{-closure}(T) = \bigcup_{s \in T} \varepsilon\text{-closure}(s) \]

  \[ \text{move}(T, a) = \{t \mid s \to_a t \text{ and } s \in T\} \]

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

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

\[
S = \epsilon\text{-closure}(s_0);
\]
\[
c = \text{nextChar}();
\]
\[
\textbf{while} ( c \neq \text{eof} ) { \}
\]
\[
S = \epsilon\text{-closure}(\text{move}(S, c));
\]
\[
c = \text{nextChar}();
\]
\[
}\textbf{if} ( S \cap F \neq \emptyset ) \textbf{return} \text{“yes”;}
\]
\[
\textbf{else return} \text{“no”;}
\]
The Subset Construction Algorithm

• 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

Initially, $\varepsilon$-closure($s_0$) is the only state in Dstates and it is unmarked

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

![ Subset Construction Example 1 Diagram ]

 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

![ Subset Construction Example 2 Diagram ]

 DSTates
 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

<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 $\varepsilon$-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 $\varepsilon$-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 $(a|b)*abb#$

- Concatenation
- Closure
- Alternation

Position number (for leaves $\neq \varepsilon$)
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

<table>
<thead>
<tr>
<th>Node $n$</th>
<th>nullable($n$)</th>
<th>firstpos($n$)</th>
<th>lastpos($n$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leaf $\varepsilon$</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>nullable($c_1$) or nullable($c_2$)</td>
<td>firstpos($c_1$) $\cup$ firstpos($c_2$)</td>
<td>lastpos($c_1$) $\cup$ lastpos($c_2$)</td>
</tr>
<tr>
<td>$\backslash$</td>
<td>nullable($c_1$) and nullable($c_2$)</td>
<td>if nullable($c_1$) then firstpos($c_1$) $\cup$ firstpos($c_2$) else firstpos($c_1$)</td>
<td>if nullable($c_2$) then lastpos($c_1$) $\cup$ lastpos($c_2$) else lastpos($c_2$)</td>
</tr>
<tr>
<td>$*$</td>
<td>true</td>
<td>firstpos($c_1$)</td>
<td>lastpos($c_1$)</td>
</tr>
</tbody>
</table>
From Regular Expression to DFA Directly: Syntax Tree of \((ab)^*abb\#
\)

From Regular Expression to DFA Directly: \textit{followpos}

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

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

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

\textbf{while} there is an unmarked state \( T \) in \( Dstates \) \textbf{do}

\hspace{1em} \text{mark} \( T \)

\hspace{1em} \textbf{for} each input symbol \( a \in \Sigma \) \textbf{do}

\hspace{2em} let \( U \) be the set of positions that are in \( \text{followpos}(p) \)

\hspace{2em} for some position \( p \) in \( T \),

\hspace{2em} such that the symbol at position \( p \) is \( a \)

\hspace{2em} \textbf{if} \( U \) is not empty and not in \( Dstates \) \textbf{then}

\hspace{3em} add \( U \) as an unmarked state to \( Dstates \)

\hspace{2em} \textbf{end if}

\hspace{2em} \( Dtran[T,a] := U \)

\textbf{end do}

\textbf{end do}

From Regular Expression to DFA Directly: Example

<table>
<thead>
<tr>
<th>Node</th>
<th>( \text{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>

\[
\begin{array}{c}
\text{start} \\
1,2,3 \\
1,2,3,4 \\
1,2,3,4,5,6 \\
1,2,3,4,5 \\
1,2,3,4,5 \\
1,2,3,4,5 \\
1,2,3,4,5,6 \\
\end{array}
\]
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

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

```
default  base
  q
```

```
next    check
  r  t
```

"a"