Organizing the Lexical Analyser

Many token types, 1 FA per token
Order them (how?) and simulate each FA in turn
Want longest possible tokens  \textbf{ex. if ifid  ex. 1 12}
When recognize token, take action (e.g. put in Symbol Table)

\begin{align*}
\text{tokenssofar} & \downarrow \text{token of total token} \\
\text{Look for next token, following edges of current FA} \\
\text{If current FA does not recognize, then go to next, reset ptr.} \\
\text{If no token recognized, then lexical error} \\
\textbf{Qu: What about separator characters (eg blanks)?} \\
\text{Should we use NFA or DFA?}
\end{align*}

How big do NFA’s get?

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
Regular Exp. & NFA \textit{(ADD)} & \\
& States & Edges \\
\hline
a & 2 & 2 \\
\varepsilon & 2 & 2 \\
\phi & 2 & 1 \\
\hline
R1 \cup R2 & 2 & 4 \\
R1 \circ R2 & 0 & 1 \\
R1 \ast & 2 & 3 \\
\hline
\end{tabular}
\end{table}

\begin{align*}
\text{has } r \text{ symbols} & \quad \text{has } \leq 6^r \ast r \text{ states} \\
\text{Can be constructed in } & \quad \text{O}(r) \text{ steps}
\end{align*}
### Regular languages are closed under ...

Th. The class of reg. lang. are closed under $\cup$

Construct NFA $N$ such that $L(N) = L(N_1) \cup L(N_2)$

Proof Idea:

![Diagram](image1)

*Start out with only 1 final state in each, end up 1 final state*

Th. The class of reg. lang. are closed under $\varepsilon$

![Diagram](image2)

### Regular languages are closed under Star

Proof Idea:

![Diagram](image3)

*Start out with 1 final state*

*End up with 1 final state*

### How big do DFA’s get?

<table>
<thead>
<tr>
<th>Reg. Exp.</th>
<th>NFA</th>
<th>DFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r$</td>
<td>$O(r)$</td>
<td>$O(2^r)$</td>
</tr>
</tbody>
</table>
Simulating a DFA

Start state $s_0$, final states $F$

$s = s_0$
$a = \text{nextchar}$

while ($a \neq \text{eof}$) do
    $s = \text{move}(S,a)$
    $a = \text{nextchar}$
    if $s$ in $F$ then return("accept") else
    return("reject")

How many steps, in size of input $i$, size of DFA?

---

Simulating an NFA

Idea: Simulate NFA by keeping track of subset of states at run-time

Start state $q_0$, final states $F$

$S = \{ \text{all states can get to from } q_0, \text{ using } \epsilon \text{ edges} \}$
$a = \text{nextchar}$

while ($a \neq \text{eof}$) do
    new$S = \text{move}(S,a)$
    $S = \{ \text{all states can get to from some state in new$S$, using 0 or more } \epsilon \text{ edges} \}$
    $a = \text{nextchar}$
    if $S \cap F \neq \phi$ then return("accept") else
    return("reject")

How many steps in terms of input $i$, size of NFA?
### Use NFA or DFA?

<table>
<thead>
<tr>
<th>Input size i</th>
<th>Size of FA</th>
<th>Simulation Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reg Exp NFA</td>
<td>$O(r)$</td>
<td>$O(r \cdot i)$</td>
</tr>
<tr>
<td>size r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reg Exp NFA</td>
<td>$O(2^r)$</td>
<td>$O(i)$</td>
</tr>
<tr>
<td>NFA DFA</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Which one to use?

- **input short**  NFA
  - Ex. Search for short pattern in text editor
- **input long**   DFA
  - Ex. Search long pattern in pattern-match program

### Tools that Use Reg. Exp and FA

<table>
<thead>
<tr>
<th>awk</th>
<th>DFA, NFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>grep</td>
<td>DFA, NFA</td>
</tr>
<tr>
<td>lex</td>
<td>DFA, NFA</td>
</tr>
<tr>
<td>emacs</td>
<td>NFA</td>
</tr>
<tr>
<td>vi</td>
<td>NFA</td>
</tr>
<tr>
<td>flex</td>
<td>NFA</td>
</tr>
<tr>
<td>Perl</td>
<td>NFA</td>
</tr>
<tr>
<td>Python</td>
<td>NFA</td>
</tr>
<tr>
<td>Tcl</td>
<td>NFA</td>
</tr>
</tbody>
</table>
Why Use Grammars?

Precise specification of language structure
   Ex. Begin-End hierarchy

Certain classes of grammars have efficient
   automatic tools to construct parsers for languages
   Ex. Bison

Provides feedback to language designer
   See if new construct difficult to handle

Adaptable
   Easy to add new language constructs

Front End of Compiler

source

Lexer

Token Stream

Parser

Parse Tree

Rest of Compiler

Main Goal of Parser: Produce parse tree via grammar
Example: Context-free Grammar

| Rules       | E → a                           |
|            | E → EAE                         |
|            | A → +                           |
|            | A → *                           |
| Terminals  | + * a                           |
| Variables  | E A                             |
| Start Variable | E                      |

Context-free since can apply variable in any context.
Use rules starting from start variable to derive strings

Using Context-free Grammar

E → EAE → aAE → a+E → a+aEAE → a+aAE → a+a*E → a+a*a

Derivation of terminal string a+a*a

Can abbreviate the rules:

E → a          written as       E → a | EAE
E → EAE

Can derive string of variables and terminals

Language of grammar is the set of terminal strings it can derive