<table>
<thead>
<tr>
<th>Regular Expressions &amp; Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular expressions are a formalism for describing patterns in strings <em>(used in tools like Emacs, Perl,...)</em></td>
</tr>
<tr>
<td>Language of a regular expression is the set of strings described</td>
</tr>
<tr>
<td>a in Σ</td>
</tr>
<tr>
<td>ε</td>
</tr>
<tr>
<td>φ</td>
</tr>
<tr>
<td>(R1 ∪ R2) (R1, R2 reg. exp)</td>
</tr>
<tr>
<td>(R1 o R2)</td>
</tr>
<tr>
<td>(R1)*</td>
</tr>
</tbody>
</table>

*What kind of definition is this?*

<table>
<thead>
<tr>
<th>Regular Expressions, Examples over {a,b}</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a ∪ b)</td>
</tr>
</tbody>
</table>

All these parentheses are clumsy—can omit but then must know what order to carry out operations

```
  a ∪ b * ???
```

Precedence of operations:

```
  *  
  o  
  (associate to left)  
  ∪
```

but ( ) must always be followed if included!

Shorthand: use Σ to stand alphabet, ex. a ∪ b

Omit o where understood
```
  (Σ Σ Σ)^* Σ^* bb Σ^* a^* b a^* φ a^* ε
```
Lexical Analysis (Scanning) in Compilers

First phase of the compiler: strings → tokens

Programming-language dependent

Tokens can include
- keywords: if, then, end
- operators: <, >, =
- identifiers
- literals: decimal integer, character

One regular expression to describe each of the token types

Qu: How recognize tokens?
   Can use NFA or DFA

What is the Regular Expression?

A **Java identifier** must start with a letter, underscore, or $; subsequent characters can also contain digits (Letters are A-Z, a-z, digits are 0-9)

A **Java decimal integer literal** consists of a sequence of digits, without a leading zero, and with an optional L or l (indicating it is a long integer)
   It must have at least one digit.

A **Java floating point literal** can have the following parts:
   a decimal integer literal, a decimal point, a fraction (decimal integer literal), an exponent (E or e followed by an optional sign and a decimal integer)
   It must have either . or an exponent.
Lex and Flex

Automatic tools for creating a lexical analyzer
Based on regular expressions

Lex source
lex.l
reg. ex. + actions

Lex Compiler

C code
lex.yy.c
tabular rep. of reg. ex.
+ table-driven routines

C Compiler

C

a.out
Lexical Analyzer executable

input stream

Lex. Anal.
a.out

sequence of tokens

Lex and Flex, Continued

Lex language

Definitions

Regular Definitions

Ex. DIGIT [0-9]
ID [a-z][a-z 0-9]*

%%

Rules

Pattern {Action}

Extended Reg. Exp. C code

0 or more blanks \[t]* putchar(' ');

0 or 1 a [a?]

not A-Z \[^A-Z]

%%

User Code C code
Regular expressions and languages

Th.: A language is regular iff some regular expression describes it

Lemma: If language $A$ is described by some reg. exp., then there is a NFA that recognizes $A$

Proof:

<table>
<thead>
<tr>
<th>Reg. Exp.</th>
<th>NFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>[Diagram of NFA for $a$]</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>[Diagram of NFA for $\varepsilon$]</td>
</tr>
<tr>
<td>$\phi$</td>
<td>[Diagram of NFA for $\phi$]</td>
</tr>
<tr>
<td>$R_1 \cup R_2$</td>
<td>[Diagram of NFA for $R_1 \cup R_2$]</td>
</tr>
<tr>
<td>$R_1 \circ R_2$</td>
<td>[Diagram of NFA for $R_1 \circ R_2$]</td>
</tr>
<tr>
<td>$R_1^*$</td>
<td>[Diagram of NFA for $R_1^*$]</td>
</tr>
</tbody>
</table>

Ex. $\{+, \varepsilon, -\} \{(0-9)\}^*$

---

DFA to Regular Expression

Lemma: If language $L$ is accepted by a DFA, then there is a reg. exp. that describes it

Proof Idea: DFA $\rightarrow$ NFA with reg exp labels $\rightarrow$ reg. exp.

Example: 1.35 p. 75

[Diagram of DFA and NFA for regular expression conversion]
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 \textbf{ex. 1 12}}
When recognize token, take action (e.g. put in Symbol Table)
\begin{align*}
\text{tokenssofar} &\leftarrow \text{new part of 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}
\end{align*}
Qu: What about separator characters (eg blanks)?
\begin{itemize}
  \item Should we use NFA or DFA?
\end{itemize}

Simulating a DFA

Start state $s0$, final states $F$
\begin{align*}
  s &\leftarrow s0 \\
  a &\leftarrow \text{nextchar} \\
  \text{while (a .neq. eof) do} \\
  \quad s &\leftarrow \text{move}(S,a) \\
  \quad a &\leftarrow \text{nextchar} \\
  \text{if } s \in F \text{ then return("accept") else} \\
  \quad \text{return("reject")}
\end{align*}

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 q0, final states F
S = \{ all states can get to from q0, using \( \epsilon \) edges \}
a = nextchar
while (a \neq \text{eof}) do
    newS = move(S, a)
    S = \{ all states can get to from some state in newS, using 0 or more \( \epsilon \) edges \}
a = nextchar
if S \cap F \neq \emptyset 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 \rightarrow NFA</td>
<td>O(r)</td>
<td>O(r \cdot i)</td>
</tr>
<tr>
<td>Reg Exp \rightarrow NFA \rightarrow DFA</td>
<td>O(2^r)</td>
<td>O(i)</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