## Regular Expressions & Languages

**Regular expressions are a formalism for describing patterns in strings** ***(used in tools like Emacs, Perl,...)***

**Language of a regular expression is the set of strings described**

| | | |
|---|---|---|
| **a** | **a in** $\Sigma$ | **{a}** |
| $\varepsilon$ | | **{**$\varepsilon$**}** |
| $\phi$ | | $\phi$ |
| **(R1** $\cup$ **R2)** | **(R1, R2 reg. exp)** | **L(R1)** $\cup$ **L(R2)** |
| **(R1**$\circ$ **R2)** | | **L(R1)** $\circ$ **L(R2)** |
| **(R1)*** | | **(L(R1))*** |

*What kind of definition is this?*



## Regular Expressions, Examples over {a,b}

**(a** $\cup$ **b)** **(a)*** **(a** $\cup$ **b)***

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

**a**$\cup$**b*** **???**

**Precedence of operations:** *(associate to left)*

*****
$\circ$
$\cup$

**but ( ) must always be followed if included!**

**Shorthand: use** $\Sigma$ **to stand alphabet, ex. a**$\cup$ **b**

**Omit** $\circ$ **where understood**

**(**$\Sigma$ $\Sigma$ $\Sigma$**)*** $\Sigma^*$**bb**$\Sigma^*$ **a*b** **a*** $\phi$ **a***$\varepsilon$

## 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.**

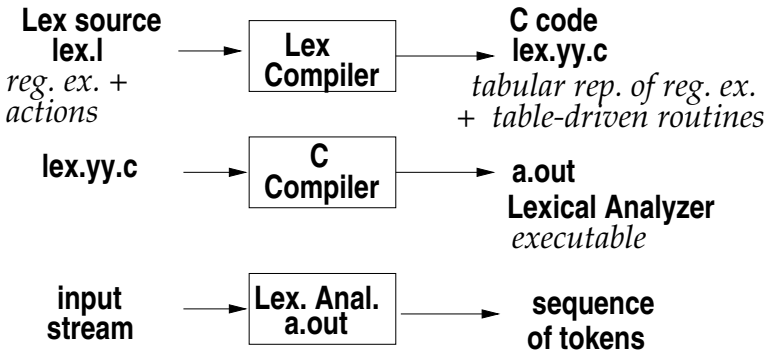
**A** *Java floating point literal* **can have the following parts: a decimal integer literal, a decimal point, a fraction (decimal integer literal), an exponent (E or e followed by an optional sign and a decimal integer)**

**It must have either . or an exponent.**

## Lex and Flex

**Automatic tools for creating a lexical analyzer**

**Based on regular expressions**






## Lex 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]**

*%%*

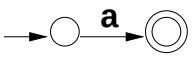
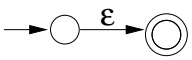
*User Code* *C code*

## 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:**

| Reg. Exp. | NFA |
|---|---|
| **a** |  |
| ε |  |
| ϕ | |
| **R1 ∪ R2** | |
| **R1 ∘ R2** | |
| **R1*** | |

**Ex. {+, ε , -} ( {0-9}) ***

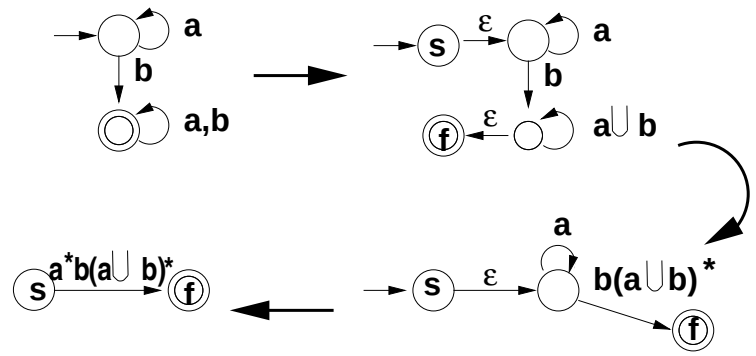


## DFA to Regular Expression

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

*Proof Idea: DFA → NFA with reg exp labels → reg. ex.*

**Example:** *1.35 p. 75*



## Organizing the Lexical Analyser

*Many token types, 1 FA per token*

*Order them (how?) and simulate each FA in turn*

*Want longest possible tokens* **ex. if ifid ex. 1 12**

*When recognize token, take action (e.g. put in Symbol Table)*

**tokenssofar***newpartof token*

*Look for next token, following edges of current FA*

*If current FA does not recognize, then go to next, reset ptr.*

*If no token recognized, then lexical error*

**Qu: What about separator characters (eg blanks)?**

**Should we use NFA or DFA?**



## Simulating a DFA

**Start state s0, final states F**

```
s = s0
a = nextchar

while (a .neq. eof) do

        s = move(S,a)
        a = 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 q0, final states F
S = {all states can get to from q0, using ε edges}
a = nextchar
while (a .neq. eof) do
        newS = move(S,a)
        S  = { all states can get to from some state
               in newS, using 0 or more ε edges}
        a = nextchar
if S ⋂ F .neq. ϕ then return("accept") else
                      return("reject")
```

*How many steps in terms of input i, size of NFA?*



## Use NFA or DFA?

| Input size i | *Size of FA* | *Simulation Time* |
|---|---|---|
| *Reg Exp→ NFA* size r | O(r) | O(r i) |
| *Reg Exp→ NFA→DFA* | $O(2^r)$ | O(i) |

**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**