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?

How big do NFA's get?

Regular Exp.	**NFA** *(ADD)* *States*	*Edges*
a	**2**	**2**
ε	**2**	**2**
ϕ	**2**	**1**
R1 $\cup$ **R2**	**2**	**4**
R1 $\circ$ **R2**	**0**	**1**
R1 *	**2**	**3**

has r symbols *has* $\leq$ *6* r states*

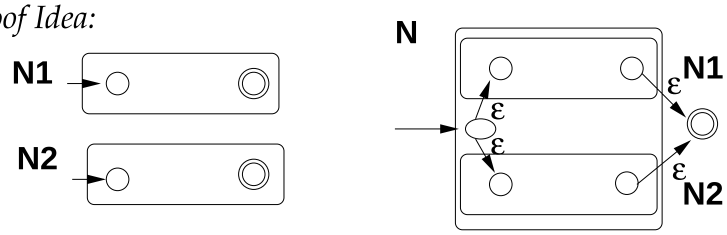
Can be constructed in O(r) steps

Regular languages are closed under ...

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

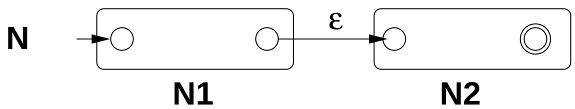
Construct NFA N such that L(N) = L(N1) $\cup$ L(N2)

Proof Idea:



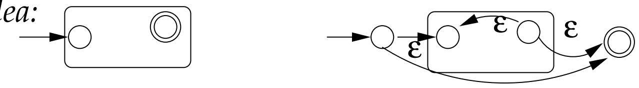
Start out with only 1 final state in each, end up 1 final state

Th. The class of reg. lang. are closed under



Regular languages are closed under Star

Proof Idea:



Start out with 1 final state

End up with 1 final state

How big do DFA's get?

Reg. Exp. $\longrightarrow$ *NFA* $\longrightarrow$ *DFA*

r **O(r)** **O(2^r)**

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

Tools that Use Reg. Exp and FA

awk	*DFA, NFA*
grep	*DFA, NFA*
lex	*DFA, NFA*
emacs	*NFA*
vi	*NFA*
flex	*NFA*
Perl	*NFA*
Python	*NFA*
Tcl	*NFA*

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

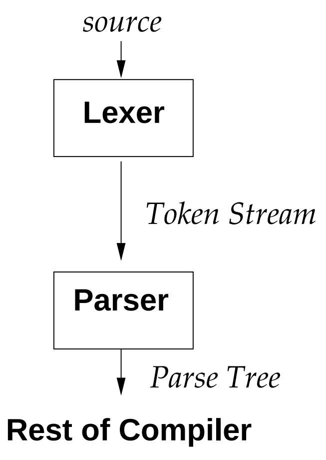
Provides feedback to language designer

See if new construct difficult to handle

Adaptable

Easy to add new language constructs

Front End of Compiler



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+EAE → a+aAE → a+a*E → a+a*a

Derivation of terminal string **a+a*a**

Can abbreviate the rules:

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

Can derive string of variables and terminals

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