**Def.** A GNFA is a 5-tuple $N = (Q, \Sigma, s, s', F)$, where now $s$ is a set of generalized instructions $(p, d, q)$, where $d$ is a regexp.

**Note.** An NFA$_\epsilon$ is the same as a GNFA in which $\epsilon$ is a basic regexp $\epsilon$ or $\epsilon$ (or $\emptyset$ for non-edges).

**Def.** A valid computation trace of $N$ is $x$ has the form $(p, U_1, q_1, U_2, q_2, U_3, \ldots, U_{m-1}, U_m, q)$, where $U_1 \ldots U_m = x$ and for each $j$, $1 \leq j \leq m$, there is an instruction $(q_{j-1}, \alpha_j, q_j)$ st. $U_j \in L(\alpha_j)$. Then we say $x \in L(pq)$, and $L(N) = \bigcup_{s \in F} U L_s$ for $s \in F$.

**Note.** For every regexp $d$ we can build the trivial equivalent GNFA $s \xrightarrow{\omega} d$ $\bigcirc$. The conversion from GNFA to regexp has this as the end case.
Examples and Practical Time-Saver: 2-stack GNN

\[ \begin{align*}
| \alpha | \quad \beta \\
q & \quad \eta
\end{align*} \]

**Concrex example:**

\[ N = \sigma s \ t \ u \ v \]

\[ L(N) = Lss \cup Lst = Lss \cdot (E + \#(0 + d)) \]

where

\[ Lss = (0 + \#(0 + d)) \]

This explains the regular expression in the "Turing Kit" DragonSt. example

\[ Lss = (\alpha \beta)^* \]

**Also:** \( (\alpha \beta)^* \equiv (\alpha \eta, \alpha \sigma) \)

**Without adding an extra start and final state, we can use this as a base case whenever the original \( N \) has at most 1 accepting state besides \( s \).**

Then, let us assume states 3, \( \ldots, k \) are nonaccepting.

**Alg:** For \( k = 2 \) down to 3:

1. Bypass and eliminate state \( k \) by:
   - For each incoming edge \((i, \beta, k)\):
   - For each outgoing edge \((k, \eta, j)\):
   - Update any existing edge \((i, \sigma, j)\)

2. \( (\alpha \beta \eta)^* \eta \) when \((k, y, k)\) is the loop at \( k \)
Extra: In UNIX, α? = α+ε. Best used only with single chars and other char exprs, eg C.

Or [a-z]? for an optional lowercase letter.

IMHO the precedence of ? is hard to remember — whereas the precedence of +, *, is the same as for plus, times, and exponents in math.

Note how this lecture showed nested *s as in (b*a + a*a(b+a)).
The combination of allowing ∼ as an extra operator on nested-* parts is what can blow up the running time of egrep: extended regular expression. But if negation is limited to char level (as in [^a-z]) for not a lowercase char, then it doesn’t blow up. Several NP-hard problems become tractable when a parameter related to allowable nesting is held to a fixed bound.

This also shows why although the regular languages are closed under ∼, we hesitate to allow it as a basic operation. Our proofs are written for U, *, and *, but not for ∼ (and sometimes not for ∨ either).

**Picture**

\[ \alpha + \beta \gamma \eta \]

\[ \gamma = \emptyset \text{ possible} \]

Then \( \gamma^* = \emptyset \).

Since \( \emptyset^* = \emptyset \), \( \beta \neq \emptyset, \eta \neq \emptyset \).

Once all out edges are null, delete all incoming edges on this state K.

Since K is a non-accepting state, any processing that goes in to K from some not-i must come out at some node j, having matched α. It could equivalently match \( \alpha + \beta \gamma \eta \) directly.
In practice, can shortcut examples.

**Example 3:**

**Inc.:** \((1, 2, 3), (2, 6, 3)\)

**Out.:** \((3, 4, 2)\)

1. \(a \rightarrow b \rightarrow c \rightarrow 2a\) 
   - Update: \((1, -1, 2), (2, -2, 2)\)

2. \((1-2)_{\text{new}} = (1-2)_{\text{old}} + (1-3)(3-3)(3-2)\)
   - \(= b^* a^* a\)

3. \((2-2)_{\text{new}} = (2-2)_{\text{old}} + (2-5)(3-3)(3-2)\)
   - \(= \emptyset + b a^* a\)

- **Final answer:** \(L_{12} = a^* (b + a^* a) \cdot L_{22}\)

- **Note:** An extra \(\emptyset\) inside the outer \(\emptyset\) would not change \(L_{22}\) and the final answer.