## PART 2. REGULAR LANGUAGES, GRAMMARS AND AUTOMATA

## RIGHT LINEAR LANGUAGES.

**Right Linear Grammar:** Rules of the form:

$A \rightarrow \alpha B,\ A \rightarrow \alpha \qquad A,B \in V_N,\ \alpha \in V_T^+$

**Left Linear Grammar:** Rules of the form:

$A \rightarrow B\alpha,\ A \rightarrow \alpha \qquad A,B \in V_N,\ \alpha \in V_T^+$

Rewrite a nonterminal into a **non-empty string of terminal symbols** or a non-empty string of terminal symbols followed by a nonterminal (right linear)/ a nonterminal followed by a non-empty string of terminal symbols (left linear).

**Restricted Right Linear Grammar:** Rules of the form:

$A \rightarrow aB,\ A \rightarrow a \qquad A,B \in V_N,\ a \in V_T$

**Restricted Left Linear:** Rules of the form:

$A \rightarrow Ba,\ A \rightarrow a \qquad A,B \in V_N,\ a \in V_T$

Rewrite a nonterminal into a **terminal symbol** or a terminal symbol followed by a nonterminal (right linear)/a non terminal followed by a terminal symbol (left linear)

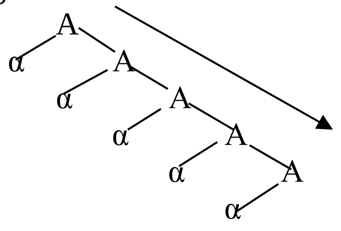
**Linear:** $A \rightarrow \alpha B \beta$, $A,B \in V_N$, $\alpha \in V_T^+$ and $\beta \in V_T^*$ or $\alpha \in V_T^*$ and $\beta \in V_T^+$
or $A \rightarrow \alpha$, $A \in V_N$, $\alpha \in V_T^+$.

Rewrite a non-terminal into a non-empty string of terminal symbols, or into a non-terminal **flanked by** two strings of terminals (possibly empty).

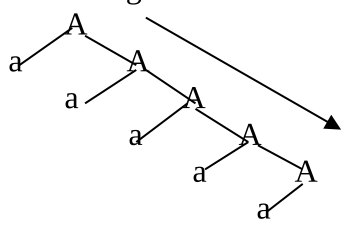
**Remark 1:** for all these type: we include in the type $G_e$ for grammars G of the type in reduced form.
**Remark 2:** Clearly, a restricted right linear grammar is right linear, a right linear grammar is linear, a linear grammar is context free.
**Remark 3:** Linear grammars are stronger than right linear grammars. We will prove shortly that language $a^nb^n$ (n>0) (all string consisting of a's followed by an equal number of b's, at least 1 a and at least 1 b) cannot be generated by a right linear grammar.
But $S \rightarrow ab$, $S \rightarrow aSb$ is a linear grammar and generates $a^nb^n$ (n>0).

The terminology is explained by the generated parse trees:

Right linear:

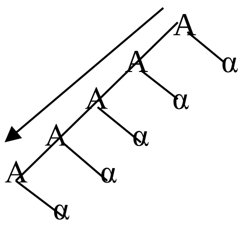


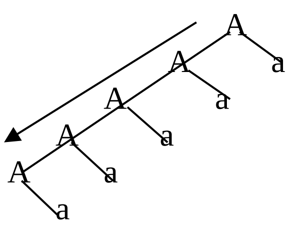
Restricted right linear



A single spine of non-terminals on the right side.

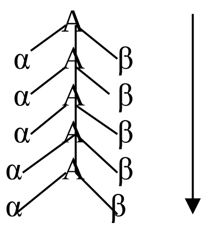
Left linear: Restricted left linear





A single spine of non-terminals on the left side.

Linear:



A single spine of non-terminals.

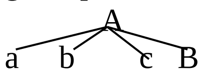
**Theorem:** For every right linear grammar there is an equivalent restricted right linear grammar.

**Proof:**
Let G be a right linear grammar. The rules are of the form:
$A \to \alpha$, $A \to \alpha B$, where $\alpha = a_1 \ldots a_n$.

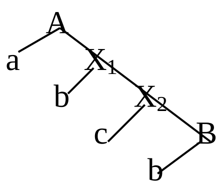
Take any such rule R. Add to the grammar **new** non-terminal symbols $X_1 \ldots X_{n-1}$ and replace R by the rules:

$$A \to a_1 X_1,\ X_1 \to a_2 X_2, \ldots \begin{cases} X_{n-1} \to a_n \\ X_{n-1} \to a_n B \end{cases} \qquad \text{depending on R}$$

Example:
$A \to abcB$, $B \to b$
gives parse tree:



$A \to aX_1$, $X_1 \to bX_2$, $X_2 \to cB$, $B \to b$
gives parse tree:

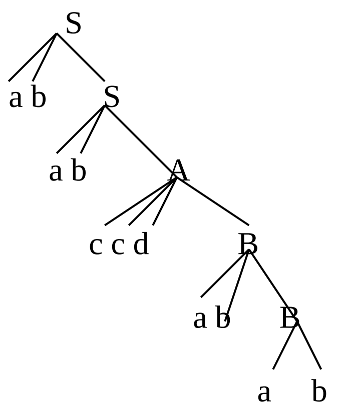


The resulting grammar generates the same languages and is a restricted right linear grammar.

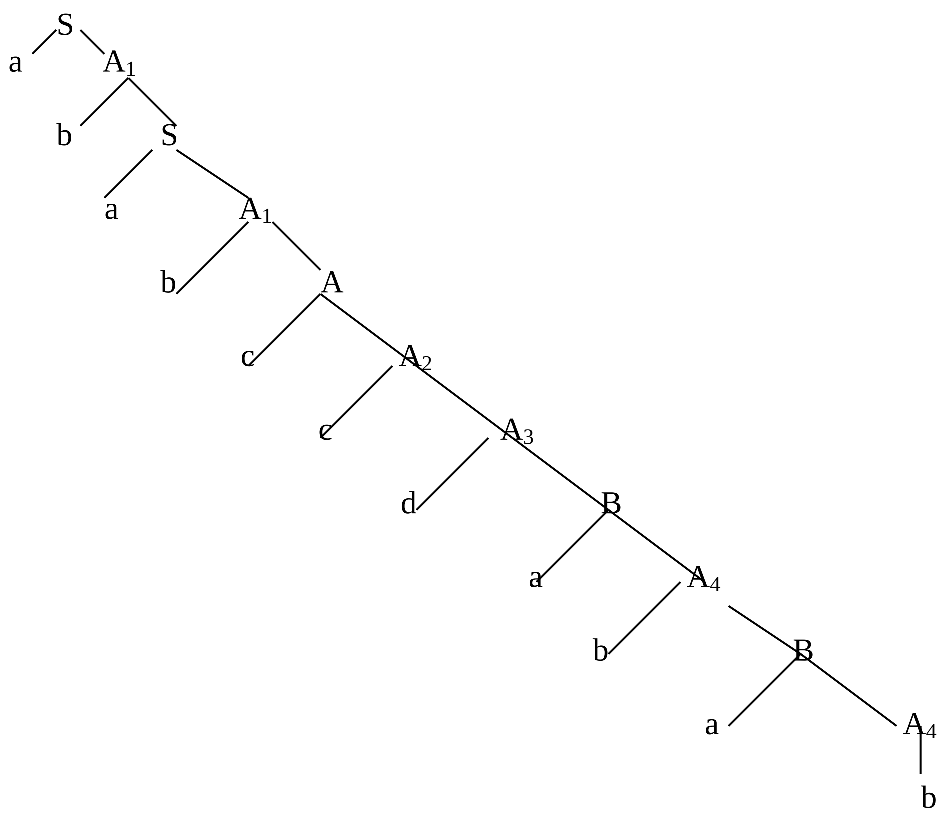
**Example:**

$(ab)^+ccd(ab)^+$

$S \rightarrow ab\ S$
$S \rightarrow ab\ A$
$A \rightarrow ccd\ B$
$B \rightarrow ab$
$B \rightarrow ab\ B$



$S \rightarrow a\ A_1$
$A_1 \rightarrow b\ S$
$A_1 \rightarrow b\ A$
$A \rightarrow c\ A_2$
$A_2 \rightarrow c\ A_3$
$A_3 \rightarrow d\ B$
$B \rightarrow a\ A_4$
$A_4 \rightarrow b$
$A_4 \rightarrow b\ A$



We obviously have the same result for left linear grammars and restricted left linear grammars.

**Theorem:** For every right linear grammar there is an equivalent left linear grammar (and vice versa).
**Proof:** This will follow from other facts later.

This means that the right linear languages are exactly the restricted right linear languages and are exactly the left linear languages.

**REGULAR LANGUAGES.**

**Kleene closure,** *, is an operation which maps every language A onto A*.
**Union,** $\cup$, is an operation which maps every two languages A and B onto $A \cup B$.
We introduce a third operation **product**: $\times$
**Product**, $\times$, is a operation which maps every two languages A and B onto $A \times B$, defined as:

The **product of** A and B: $A \times B = \{\alpha^\wedge\beta: \alpha \in A \text{ and } \beta \in B\}$

Example: If A = {a,b} and B = {cc, d}
$A \times B =$ {acc, ad, bcc, bd}

**Fact**: $A^+ = A \times A^*$
**Proof:**
Case 1. Let $e \in A$.
Then $A^+ = A^*$ , so the claim is: $A^* = A \times A^*$
-Let $\alpha \in A$ and $\beta \in A^*$, then obviously, by definition of $A^*$, $\alpha^\wedge\beta \in A^*$, hence $A \times A^* \subseteq A^*$.
-$A \times A^* = \{\alpha^\wedge\beta\text{: } \alpha \in A \text{ and } \beta \in A^*\}$
$A^* = \{e^\wedge\beta\text{: } \beta \in A^*\}$, and since $e \in A$, $A^* \subseteq A \times A^*$.
So indeed $A \times A^* = A^*$.

Case 2. $e \notin A$.
-We have already proved that $A \times A^* \subseteq A^*$.
-Since $e \notin A$, $e \notin A \times A^*$, because every string in $A \times A^*$ starts with as string from A.
The two prove that $A \times A^* \subseteq A^+$.
-Let $\alpha \in A^+$.
Then either $\alpha \in A$, and since $\alpha = \alpha^\wedge e$ and $e \in A^*$, $\alpha \in A \times A^*$.
Or $\alpha = \alpha_1{}^\wedge...{}^\wedge\alpha_n$, where $\alpha_1,...,\alpha_n \in A$. But then $\alpha_1 \in A$ and $\alpha_2{}^\wedge...{}^\wedge\alpha_n \in A^*$.
Hence, $\alpha \in A \times A^*$. So, $A^+ \subseteq A \times A^*$.
Consequently, $A^+ = A \times A^*$.

**Example:**
$A = \{a,b\}$
$A^* = \{e, a, b, ab, ba, aa, bb, aaa, \dots\}$
$A \times A^* =$
$\{ a^\wedge e$, aa, ab, aab, aba, aaa, abb, aaaa,….
$b^\wedge e$, ba, bb, bab,…} =
{a, b, ab, ba, aa, bb, aaa, ….} = $A^+$

Take two languages A and B. First take their union, $(A \cup B)$. Then take the Kleene closure of that: $(A \cup B)^*$.
The **composition** operation $* \circ \cup$ is the operation which takes any two languages A and B and maps them onto $*(A \cup B)$.

Let O be the operation which maps any five languages A,B,C,D,E onto the language $((A)^*\cup(B)^*)\times(C\times(D\cup E)^*)^*)$. This operation can be **decomposed as a finite sequence of compositions of the operations** $*,\cup,\times$:
First apply * to A , then apply * to B, then apply $\cup$ to the result. Call this 1.
Then apply union to D and E and apply * to the result. Call this 2.
Apply $\times$ to C and 2, and apply * to the result. Call this 3.
Now apply $\times$ to 1 and 3 and you get the output of O.

Defining the notion of 'finite sequence of compositions' is technically complex and nitty-gritty. I won't do that here, but assume instead that the intuition is clear.

The class of **regular operations** on languages is given by:

1. $*, \cup, \times$ are regular operations on languages.
2. Any operation which can be decomposed as a finite sequence of compositions of the operations $*, \cup, \times$ is a regular operation.

We define:

Language A is a **regular** language iff
A is a finite language or there is regular operation O and finite languages $A_1,\dots,A_n$ and $A = O(A_1,\dots,A_n)$.

This means that any regular language can be gotten by starting with a finite number of finite languages and applying a finite sequence of the operations $*, \cup, \times$.

For example, $a^n b^m (n,m \geq 0)$ is: $\{a\}^* \times \{b\}^*$
$a^n b^m (n,m \geq 1)$ is: $\{a\}^+ \times \{b\}^+$
Since we have shown that $\{a\}^+ = \{a\} \times \{a\}^*$, we see that:
$a^n b^m (n,m \geq 1)$ is: $(\{a\} \times \{a\}^*) \times (\{b\} \times \{b\}^*)$

Equivalently, we can define the class of regular languages inductively as:

*R*, the class of all **regular languages** is the smallest class such that:

1. Every **finite** language is regular.
2. If A and B are regular languages, then $A \cup B$ is regular.
3. If A and B are regular languages, then $A \times B$ is regular.
4. If A is a regular language, then $A^*$ is regular.

(We say 'class' and not 'set' because in this definition we don't put any constraints on the alphabets that the languages are languages in.)

**Theorem:** Every regular languages is a right linear language.,

**Proof**:
**Step 1**: Every finite language is a right linear language.
Let $A = \{\alpha_1,\dots,\alpha_n\}$
$S \rightarrow \alpha_1, \dots, S \rightarrow \alpha_n$ is a right linear grammar.
Note that this also holds if $e \in A$, because this grammar is trivially in reduced form.

**Step 2:** If A and B are right linear languages, then $A \cup B$ is a right linear language.
Suppose $G_A$ is a right linear grammar generating A and $G_B$ is a right linear grammar generating B.

-Change everywhere every non-terminal X in $G_A$ by a new non-terminal $X_A$.
-Change everywhere every non-terminal X in $G_B$ by a new non-terminal $X_B$.
(i.e. we make all non-terminals in $G_A$ and $G_B$ disjoint).
-Take the union of the resulting grammars.

-For every rule of the form:

$S_A \to \alpha\, A_A$, $S_A \to \alpha$, $S_B \to \beta\, B_B$, $S_B \to \beta$

**add** a rule of the form:

$S \to \alpha\, A_A$, $S \to \alpha$, $S \to \beta\, B_B$, $S \to \beta$

Call the resulting grammar $G_{A\cup B}$.

(Note that $G_{A\cup B}$ is in reduced form.)

$G_{A\cup B}$ is a right linear grammar and $G_{A\cup B}$ generates $A\cup B$.

**Step 3**: If A and B are right linear languages, then $A \times B$ is a right linear language.
Suppose $G_A$ is a right linear grammar generating A and $G_B$ is a right linear grammar generating B.

-Change everywhere every non-terminal X in $G_B$ by a new non-terminal $X_B$ (not occurring in $G_A$, i.e. we make all non-terminals in $G_A$ and $G_B$ disjoint).

1. If $e \notin A$ and $e \notin B$, then

**replace** every $G_A$ rule of the form:

$A \to \alpha$

by a rule of the form:

$A \to \alpha\, S_B$

Call the resulting grammar $G_{A\times B}$

2. If $e \notin A$ and $e \in B$, $G_B$ contains rule $S_B \to e$.

**Delete** that rule and **add** for every $G_A$ rule of the form:

$A \to \alpha$

a rule of the form:

$A \to \alpha\, S_B$

Call the resulting grammar $G_{A\times B}$

3. If $e \in A$ and $e \notin B$, then

**delete** $S \to e$ and

**replace** every remaining $G_A$ rule of the form:

$A \to \alpha$

by a rule of the form:

$A \to \alpha\, S_B$

and **add** for every rule of the form:

$S_B \to \alpha\, B_B$ or $S_B \to \alpha$

a rule of the form:

$S \to \alpha\, B_B$ or $S \to \alpha$

Call the resulting grammar $G_{A\times B}$

4. If $e \in A$ and $e \in B$, then $G_B$ contains rule $S_B \to e$.

**Delete** that rule and **add** for every remaining $G_A$ rule of the form:

$A \to \alpha$

a rule of the form:

$A \to \alpha\, S_B$

and **add** for every rule of the form:

$S_B \rightarrow \alpha B_B$ or $S_B \rightarrow \alpha$

a rule of the form:

$S \rightarrow \alpha B_B$ or $S \rightarrow \alpha$

Call the resulting grammar $G_{A \times B}$

In all four cases $G_{A \times B}$ is a right linear grammar and $G_{A \times B}$ generates $A \times B$.

**Step 4**. If A is a right linear language, then A* is a right linear language.
Suppose $G_A$ is a right linear grammar generating A.
-If $G_A$ contains rule $S \rightarrow e$, delete that rule.
-For every remaining rule of the form:

$A \rightarrow \alpha$

**add** a rule:

$A \rightarrow \alpha S$

This will generate $A^* - \{e\}$.
-Convert the resulting grammar into reduced form and add $S \rightarrow e$ to the result.
Call the result $G_{A^*}$.
$G_{A^*}$ is a right linear grammar and generates A*.
This completes the proof.

So we know now that the class of regular languages is a subclass of the class of right linear languages. We will soon see that the two classes actually coincide.

## FINITE STATE AUTOMATA

We now take a **parsing** perspective. Whereas grammars generate strings, **automata** read strings symbol by symbol, from left to right, follow instructions, and determinine, when the string is read, whether the string is **accepted or rejected**.

At any point of its operation, we assume that the automaton is in a certain state. We can think of this state as an array of switches which can be on or off. Each combination of switch-settings that the automation allows is a possible state that the automaton can be in. The instructions that the automaton follows, then, can be interpreted as instructions to reset switches, and hence as instructions to move from one state to another.

A **state automaton** is an automaton that can do this and not more: it can read the input from left to right, symbol by symbol, and at each point follow an instruction to switch state. The state that it is in after reading the input will determine whether or not the string is accepted. Importantly:

**A state automaton does not have any memory.**

A **finite state automaton** is a state automaton that has a finite number of possible states it can be in.

A finite state automaton is **deterministic** iff in any state it is in, there is **at most one** state it can switch to, according to its instructions.
A finite state automaton is **non-deterministic** iff possibly in some state there is more than one state it can switch to, according to its instructions.
A deterministic state automaton is **total** iff in any state it is in, there is **exactly one** state it can switch to.

In the formal definition we only specify the things that vary from automata to automata:

A **finite state automaton** is a tuple $M = \langle S,\Sigma,\delta,S_0,F\rangle$ where:
1. S is a **finite** set, the set of **states**.
2. $\Sigma$ is a finite **alphabet**, the **input** alphabet.
3. $\delta$, the **transition relation,** is a three-place relation relating a symbol in $\Sigma$ to two states (an input state and an output state): $\delta \subseteq S \times \Sigma \times S$.
4. $S_0 \in S$. $S_0$ is the **initial state**.
5. $F \subseteq S$. F is the set of **final states**.

Let M be a finite state automaton:
M is **deterministic** iff $\delta$ is a **partial function** from $S \times \Sigma$ into S, i.e. iff $\delta$ maps every pair consisting of an input state and an input symbol onto at most one output state.

Let M be a deterministic finite state automaton.
M is **total** iff $\delta$ is a **total function** from $S \times \Sigma$ into S, i.e. iff $\delta$ maps every pair consisting of an input state and an input symbol onto exactly one output state.

Since we read 'non-deterministic' as 'possibly non-deterministic', we take 'finite state automaton' and 'non-deterministic finite state automaton' to be the same notion.

I mentioned that we only specify the variable parts of the automaton. The invariable parts are the following:

1. Every automaton has an input tape, on which a string in the input alphabet is written.
2. Every automaton has a reading head which reads one symbol at a time.
3. Every computation **starts** while the automaton is **in the initial state $S_0$, reading the first symbol** of the input string.
4. We assume that **after** having read the last symbol of the input string, the automaton reads e.
5. At each computation step the automaton follows a transition.
We write transition $\delta(S_i,a)=S_k$ as:

$(S_i,a) \rightarrow S_k$

And with this transition, the automaton can perform the following computation step:

> **Computation step:**
> **If the automaton is in state $S_i$ and reads symbol a on the input tape, it switches to state $S_k$ and reads the next symbol on the input tape.**

6. We say:

The automaton **halts** iff there is no transition rule to continue.

Let $\alpha \in \Sigma^*$.
A **computation path for** $\alpha$ **in** M is a sequence of computation steps beginning in $S_0$ reading the first symbol of $\alpha$, following instructions in $\delta$ until M halts.

**Fact:** If M is a deterministic finite state machine, then every input string $\alpha \in \Sigma^*$ has a unique computation path.

This means that for each input string, the automaton will halt.
Now, since $e \notin \Sigma$ (since $\Sigma$ is an alphabet), there is by definition of $\delta$ no instruction if M reads e. This means that if the automaton reads e, it halts. We use this in defining acceptance:

$7_{DET}$. Deterministic finite state automaton M **accepts** string $\alpha \in \Sigma^*$ iff at the end of the computation path of $\alpha$ in M, M reads e and M is in a final state. Otherwise M **rejects** $\alpha$.

This means that M **rejects** $\alpha$ if, at the end of the computation path for $\alpha$ in M, M reads e, but is not in a final state, or if M halts at a symbol before reading the whole input string, that is, if at the end of the computation path of $\alpha$ in M, M **doesn't** read e.

If M is a non-deterministic automaton, there may be **more than one** instruction that M can follow while reading an input symbol in a state. This means that M can **choose**, and this means that for each string α of the input alphabet there may be **more than one computation path for** α **in** M. (where a computation path for α in M is, once again, a sequence of computation steps licensed by transitions in M, starting in $S_0$ reading the first symbol of α, and halting in some state.) For non-deterministic automata we define acceptance:

$7_{NDET}$ Non-deterministic finite state automaton M **accepts** string $\alpha \in \Sigma^*$ iff **for some computation path for** α **in** M, at the end of that computation path, M reads e and M is in a final state. Otherwise M **rejects** α.

This means, that there may be computation paths for α in M at the end of which M is not reading e, or M is not in a final state, **and yet** M accepts α: as long as there is **at least one computation path**, where M ends up reading e in a final state, M accepts α.

8. Let M be a finite state automaton.
L(M), the **language accepted by** M, is the set of all accepted input strings. (So $L(M) \subseteq \Sigma^*$). We call L(M) a **finite state language**.

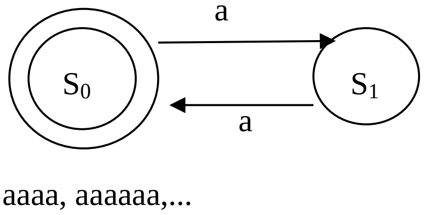
$M_1$ and $M_2$ are **equivalent** iff $L(M_1) = L(M_2)$.

Now we introduce pictures of finite state automata, called state diagrams:

A **state diagram** of a finite state automaton represents the states as circles with the state names as labels, it represents the transitions in δ as arrows between the appropriate state circles, where each arrow is labeled by the appropriate input symbol, according to δ, and it represents final states as double circles.

Example.
M is given by:

$S = \{S_0, S_1\}$
$\Sigma = \{a\}$
$\delta(S_0,a)=S_1$
$\delta(S_1,a)=S_0$
$F = \{S_0\}$

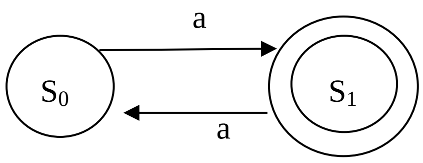


Accepted: e, aa, aaaa, aaaaaa,...
Rejected: a, aaa, aaa, aaaaa,...
Accepted language: $a^n$ (n is even.)

Note that e is accepted by this automaton.
The automaton starts out in $S_0$ reading e, and halts there. Since $S_0$ is a final state, e is accepted.
**Fact**: finite state automaton M accepts e iff $S_0$ is a final state.

M is given by:

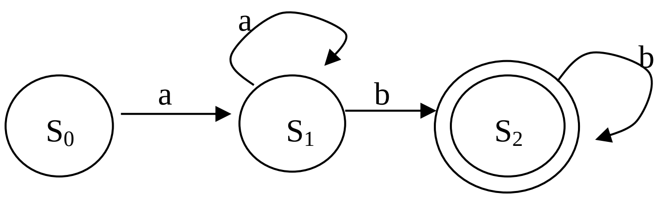
$S = \{S_0,S_1,S_2\}$
$\Sigma = \{a\}$
$\delta(S_0,a)=S_1$
$\delta(S_1,a)=S_0$
$F = \{S_1\}$



Accepted: a, aa, aaa, aaaaa, ...
Rejected: e, aa, aaaa, aaaaaa,...
Accepted language: $a^n$ (n is odd)

M is given by:

$S = \{S_0,S_1\}$
$\Sigma = \{a,b\}$
$\delta$: $(S_0,a)=S_1$
$(S_1,a)=S_1$
$(S_1,b)=S_2$
$(S_2,b)=S_2$
$F = \{S_1\}$



Accepted language: $a^nb^m$ (n>0, m>0)

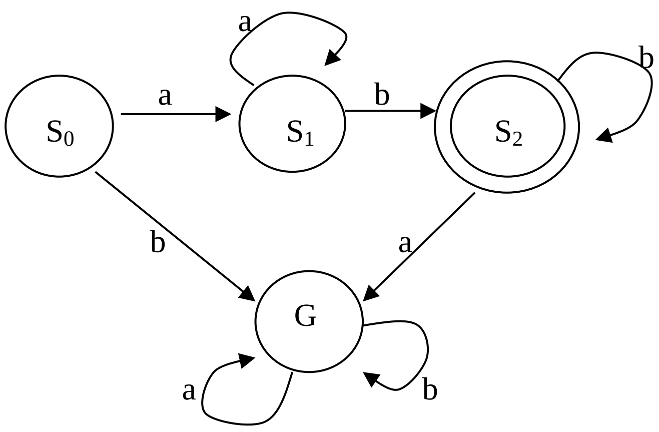
Two state diagrams are **equivalent** iff they are state diagrams for equivalent automata.
A state diagram is **reduced** iff there is no equivalent state diagram with fewer states.
**Reduction Theorem**: Any two reduced equivalent state diagrams are isomorphic (i.e. differ at most in the names of the states).
**Proof**: Omitted.

This means that for each finite state language, there is, up to isomorphism, a unique smallest automaton recognizing that language.

**Theorem:** For every deterministic finite state automaton, there is an equivalent **total** deterministic finite state automaton.

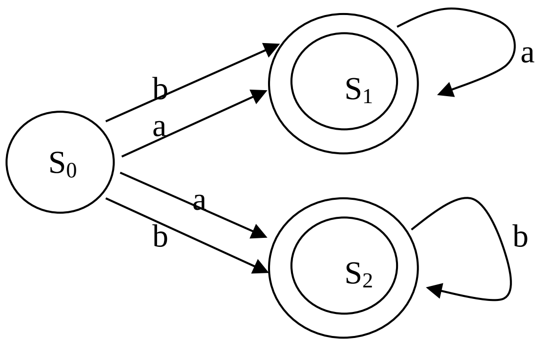
**Proof:**
Let M be a deterministic finite state automaton. Add a new state G (for garbage) to M, which is not a final state.
For each state $S_i$ and each symbol a such that $\delta(S_i,a)$ is undefined, add: $\delta(S_i,a)=G$ (also for G itself). The resulting automaton is deterministic and total, and clearly recognizes the same language as M.

Example.
We make the last automaton total:

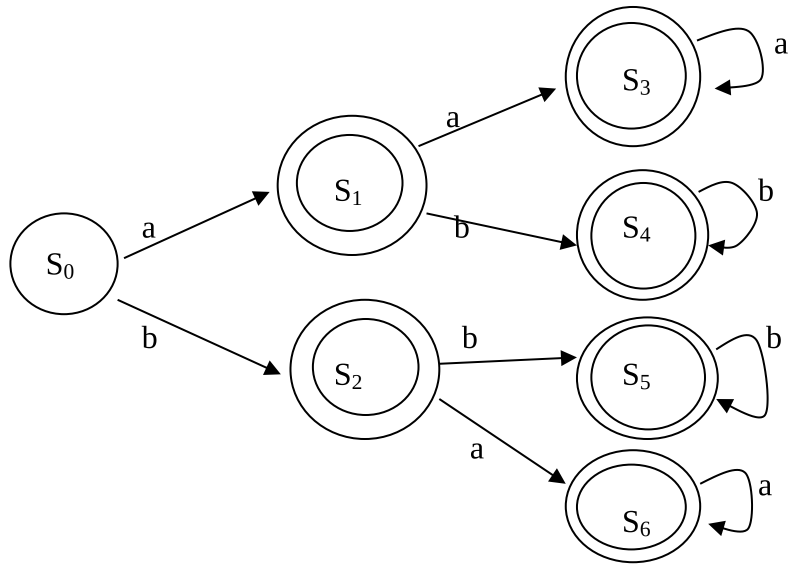


Since it is so easy to turn a deterministic automaton into a total deterministic automaton, when I ask you to make a deterministic automaton, I don't require you to make it total.
But make sure that it **is** deterministic, because often it is much easier to make a non-deterministic automaton than a deterministic one (and I sometimes don't want you to do the easier thing).

Example:
In alphabet {a,b} I want an automaton that recognizes
$a^n \cup b^m \cup ab^k \cup ba^p$ ($n,m>0,k,p \geq 0$)
-any string of one or more a's
-any string of one or more b's
-any string consisting of one b, followed by as many a's as you want.
-any string consisting of one a, followed by as many b's as you want.
This is straightforward to do non-deterministically.



On the path from $S_0$ to $S_1$ and looping on $S_1$, you accept any string of one or more a's, and any string starting with b, followed by any number of a's.
On the path from $S_)$ to $S_2$ and looping on $S_{2,}$ you accept one or more b's and any string starting with a, followed by any number of b's. Since what the automaton accepts is the union of what it accepts along each of these paths, it accepts the language specified.
Deterministically, you need to think a bit more, though it is not very difficult:

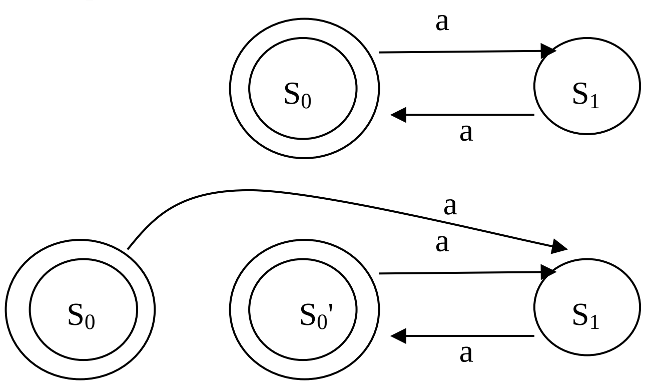


We will prove below that the class of non-deterministic finite state languages and the class of deterministic finite state languages coincide. But we will prove some simpler things first.

It is useful to introduce for automata a familiar notion and a familiar theorem:

Finite state automaton M **is in reduced form** iff $S_0$ does not occur in the range of $\delta$, i.e. if M has no arrows going into $S_0$.
**Theorem:** Every finite state automaton is equivalent to a finite state automaton in reduced form.
**Proof**: The same as for grammars: replace each occurrence of $S_0$ in M by $S_0$'' add a new initial state $S_0$, and add for each transition $(S_0',a)\rightarrow S_k$ a transition $(S,a)\rightarrow S_k$. Make $S_0$ a final state iff $e \in L(M)$. The resulting automaton is in reduced form and generates the same languages as M.

Example:

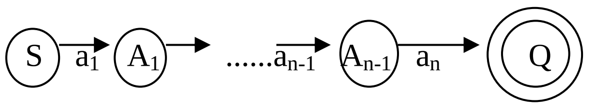


**Theorem:** The right linear languages are exactly the finite state languages.

**Proof:**
**Step 1:** If a language is right linear, there is a finite state automaton accepting it.
Let $G = \langle V_N,V_T,S,P\rangle$ be a restricted right linear grammar.

We construct a finite state automaton M:
1. $\Sigma = V_T$.
2. $S = V_N \cup \{Q\}$, with Q a symbol not in $V_N$.
3. For every rule $A \rightarrow a\ B$, we have a transition $(A,a) \rightarrow B$ in $\delta$.
4. For every rule $A \rightarrow a$, we have a transition $(A,a) \rightarrow Q$ in $\delta$.
5. S is the initial state.
6. -If $S \rightarrow e$ is not in G, then Q is the final state.
   -If $S \rightarrow e$ is in G, then Q and S are the final states.

**Claim**: G and M are equivalent.
**A.** If G generates $\alpha$, M accepts $\alpha$.
-If $\alpha$=e and G generates $\alpha$, then S is a final state and M accepts e.
-Suppose G generates $\alpha$, and $\alpha = a_1 ... a_n$.
Then there is a derivation in G of the form:
$S \Rightarrow a_1A_1 \Rightarrow .... \Rightarrow a_1...a_{n-1}A_{n-1} \Rightarrow a_1....a_n$
This means that G contains rules:
$S \rightarrow a_1A_1, ..., A_{n-2} \rightarrow a_{n-1}A_{n-1}, A_{n-1} \rightarrow a$
This means that in the automaton we have:

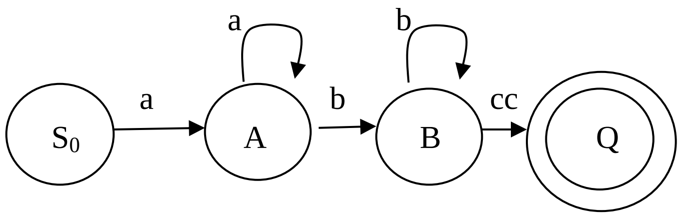


Clearly, then M accepts $\alpha$.
**B.** If M accepts $\alpha$, G generates $\alpha$.
-If $\alpha$=e and M accepts e, then $S \rightarrow e$ is in G, by definition of M, so G generates e.
-If M accepts $\alpha$ and $\alpha = a_1...a_n$, M contains a path of the above form.
Note that even though, S may in principle be a final state, if $\alpha \neq e$, M will not accept $\alpha$ in S, because S is only a final state if $S \rightarrow e$ is in G, and that can only be the case if G is in reduced form. But that means that M is also in reduced form, and this means indeed that the path accepting $\alpha$ is of the above form.

But from the construction of M, we know that then all of the rules:
$S \rightarrow a_1A_1, ..., A_{n-2} \rightarrow a_{n-1}A_{n-1}, A_{n-1} \rightarrow a$ are in G (because that's how we got those transitions in the first place). Hence G generates α.

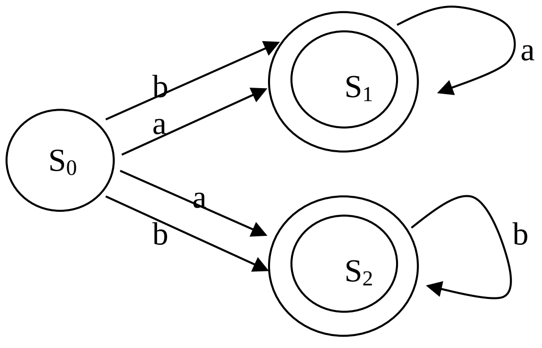
Example:

$S \rightarrow a\,A$
$A \rightarrow a\,A$
$A \rightarrow b\,A$
$B \rightarrow b\,B$
$B \rightarrow c$



**Step 2:** If a language is a finite state language, there is a right linear grammar generating it.
Let M be a finite state automaton in reduced form.
We define grammar $G_M$:
1. $V_T = \Sigma$.
2. $V_N = S$
3. For every instruction in δ: $(A_i,a) \rightarrow A_k$ we add a rule: $A_i \rightarrow aA_k$.
4. For every instruction in δ: $(A_i,a) \rightarrow F$, where F is a final state, we add a rule: $A_i \rightarrow a$.
5. $S=S_0$.
6. If $S_0$ is a final state, we add $S \rightarrow e$.
Since M was in reduced form, $G_M$ is in reduced form. Clearly, by an argument which is the inverse of the above argument, $G_M$ will generate what M accepts. And $G_M$ is right linear. Since the class of finite state languages is the class of languages accepted by finite state automata in reduced form, we have proved our theorem.

Example:



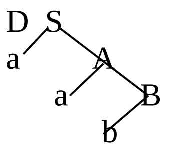
| | | | |
|---|---|---|---|
| $S \rightarrow a\,S_1$ | $S \rightarrow b\,S_1$ | $S \rightarrow a$ | $S \rightarrow b$ |
| $S \rightarrow a\,S_2$ | $S \rightarrow b\,S_2$ | | |
| $S_1 \rightarrow a\,S_1$ | | $S_1 \rightarrow a$ | |
| $S_2 \rightarrow b\,S_2$ | | $S_2 \rightarrow b$ | |

**Theorem:** The Left linear languages are exactly the finite state languages.

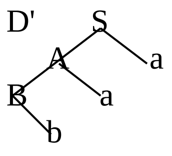
**Proof:**
We define for string $a_1...a_n$, for language A, and for restricted right linear grammar G:

The **reversal of** $a_1...a_n$, $(a_1a_2...a_n)^R = a_n...a_2a_1$
The **reversal of** A, $A^R = \{\alpha^R: \alpha \in A\}$
The **reversal of** G, $G^R$ is the result of replacing in G every rule of the form $A \rightarrow aB$ by $A \rightarrow Ba$.

**Fact**: $L(G^R) = (L(G))^R$
**Proof**: This is obvious: Right linear derivation D gets replaced by left linear derivation D':



aab



baa

**THEOREM**: If A is a finite state language, $A^R$ is a finite state language.
**Proof**:
Let M be a finite state automaton that accepts A.
Case 1. Assume M has one final state F.
-turn every transition $(S_i,a) \rightarrow S_k$ into $(S_k,a) \rightarrow S_i$.
-make $S_0$ the final state.
-make F the initial state.
The resulting finite state automaton accepts $A^R$.
Case 2. Assume M has final states $F_1,...,F_n$.
-turn every transition $(S_i,a) \rightarrow S_k$ into $(S_k,a) \rightarrow S_i$.
-make $S_0$ the final state.
-add a new initial state S', and add for every transition:
$(F_i,a) \rightarrow S_k$ a transition $(S',a) \rightarrow S_k$.
The resulting automaton is in reduced form. If $e \in A$, make S' a final state as well.
The resulting automaton recognizes $A^R$.
This completes the proof.

**Corrollary**: The left linear languages are exactly the right linear languages.
**Proof**:
-Let A be a right linear language. Then A is a finite state language. Then $A^R$ is a finite state language, by the above theorem, and hence $A^R$ is a right linear language. Take a right linear grammar G for $A^R$. $G^R$ is a left linear grammar that generates $A^{RR}$, by the earlier theorem. But $A^{RR} = A$. Hence A is a left linear language.
-Let A be a left linear language. Then $A^R$ is a right linear language, hence $A^R$ is a finite state language, hence $A^{RR}$ is a finite state language, so A is a finite state language, and hence A is a right linear language.

The next proof is a difficult proof. It is one of the two difficult proofs I do in this class. I do it, because it illuminates the structure of regular languages so well.

Remember that we proved earlier that every regular language is a right linear language, and hence (we now know) a finite state language. We will now prove the converse of this:

**THEOREM**: Every finite state language is a regular language.

**Proof**:
Let M be a finite state automaton with n states.
Assign numbers 1,...,n to the states in M: state m is the state we assign number m.

We are going to define **for each number** $k \leq n$ and **each two states** i and j, with $i,j \leq n$, **a set of string** $\mathbf{R^k_{i,j}}$.

The intuition is that we look at all the paths through the automaton that bring you from state i to state j, and we are interested in the strings that are **accepted along** those paths. This does **not** mean that these strings are accepted by the automaton M, but only that **if you start in state i**, these strings will bring you from there to state j.

The **number k** puts a **restriction** on which paths to include and which to exclude.
k says: **ignore any path that goes through any state m where m > k**.

This means, then, that $R^n_{i,j}$ is the set of **all strings** that bring you from state i to state j, because there are no states m with m > n, so all paths count.

Similarly, $R^0_{i,j}$ is the set of strings that bring you from state i to state j, while **ignoring any path** that goes through a state 1,....,n. We will interpret that as meaning that $R^0_{i,j}$ is the set of strings that **directly** bring you from state i to state j.

Following this intuition, we will define the latter sets as follows:

**Definition**: for every $i,j \leq n$:
if $i \neq j$ then: $R^0_{i,j} = \{\alpha\colon \delta(i,\alpha) = j\}$
if $i = j$ then: $R^0_{i,j} = \{\alpha\colon \delta(i,\alpha) = j\} \cup \{e\}$ (i.e. this is $R^0_{i,i}$)

We are now going to look at $R^k_{i,j}$ where $k > 0$.

$R^k_{i,j}$ is the set of strings which bring you from i to j, without going through any state with number higher than k.

Intuitively, we can split this set of strings into two sets:
-the set of strings that bring you from i to j, without going though any state with number higher than k–1: that is, $R^{k-1}_{i,j}$

-the set of strings that bring you from i to j, while going through state k.
Let us call the latter set for the moment K.
That means that:

$$R^k_{i,j} = R^{k-1}_{i,j} \cup K$$

Now we focus our attention on set K, the set of strings that bring us from i to j while going through state k.
Such strings may go through state k more than once, in that case they loop in state k.
But intuitively we can divide any such string into three parts:

-a string that brings you **from state i to state k on a path that doesn't itself go through state k. (**i.e. the string you get the first time you reach state k).
-a string that brings you **from state k to state k 0 or more times.**
-a string that brings you **from state k to state j on a path that doesn't itself go through state k** (i.e. the string you get by going from the last time you are in state k to state j).

Thus, any string in K is a **concatenation** of a string in $R^{k-1}_{i,k}$ followed (possibly) by a string that loops from k to k, followed by a string in $R^{k-1}_{k,j}$
Writing for the moment L for the set of all middle parts, the strings that loop in k, we see that:

$$K = R^{k-1}_{i,k} \times L \times R^{k-1}_{k,j}$$

Now, the loop strings are strings that bring you from state k to state k.
Each such string can be described as a concatenation of strings that bring you from state k to state k **without going through state k itself.**
This is obvious: if you loop m times in state k and get string α, divide α into the substrings you get each time you reach state k again: these substrings **themselves** do not go through state k.
This means that loop set L is the **closure under string formation** of the set $R^{k-1}_{k,k}$ the string closure of the set of strings that bring you from k back to k, without going through k (or a state with a higher number):

$$L = (R^{k-1}_{k,k})^*$$

Filling in L in K, we get:

$$K = R^{k-1}_{i,k} \times (R^{k-1}_{k,k})^* \times R^{k-1}_{k,j}$$

Filling in K in $R^{k}_{i,j}$, we get:

$$R^{k}_{i,j} = R^{k-1}_{i,j} \cup (R^{k-1}_{i,k} \times (R^{k-1}_{k,k})^* \times R^{k-1}_{k,j})$$

This we use as a definition:

**Definition:** for every k, such that $0 < k \leq n$,
for every $i,j \leq n$:

$$R^{k}_{i,j} = R^{k-1}_{i,j} \cup (R^{k-1}_{i,k} \times (R^{k-1}_{k,k})^* \times R^{k-1}_{k,j})$$

This means that, with our two definitions, we have defined $R^{k}_{i,j}$ for every number $k \leq n$, and for every states $i,j \leq n$.

Now we state a theorem:

**Theorem**: for every number $k \leq n$ and for every two states $i,j \leq n$: $R^k_{i,j}$ is regular.

**Proof:**

We prove this with induction to the number k. We will prove the following two things:

**Proposition 1:** For every two states $i,j \leq n$: $R^0_{i,j}$ is regular.
**Proposition 2:** For any number k with $0 < k \leq n$:
If it is the case that for every two states a,b: $R^{k-1}_{a,b}$ is regular,
Then it is the case that for every two states i,j: $R^k_{i,j}$ is regular

The proofs of these two propositions together form an **induction proof** of the theorem, for the following reason:
Proposition 2 says that if the theorem holds for $k-1$, it holds for k (with $k > 0$).
Since proposition 1 says that the theorem holds for $k=0$, it then follows with proposition 2, that the theorem holds for $k=1$.
It holds for $k=1$, so once again, proposition 2 says it holds for $k=2$, etc.
This means that, if we can prove propositions 1 and 2, we have indeed proved the theorem.

**Proof of proposition 1:**
By definition of $R^0_{i,j}$, $R^0_{i,j}$ is a **finite** set for every i and j, hence for every i and j, $R^0_{i,j}$ is regular (because finite sets are regular).

**Proof of proposition 2:**
We assume that it is the case for every two states a,b that $R^{k-1}_{a,b}$ is regular.
Let i and j be any states. We prove that $R^k_{i,j}$ is regular.

By definition:

$$R^k_{i,j} = R^{k-1}_{i,j} \cup (R^{k-1}_{i,k} \times (R^{k-1}_{k,k})^* \times R^{k-1}_{k,j})$$

By assumption:

$R^{k-1}_{i,j}$ is regular, $R^{k-1}_{i,k}$ is regular, $R^{k-1}_{k,k}$ is regular, and $R^{k-1}_{k,j}$ is regular.

But then $R^{k-1}_{i,j} \cup (R^{k-1}_{i,k} \times (R^{k-1}_{k,k})^* \times R^{k-1}_{k,j})$ is regular, since it is built from those sets with regular operations $\cup$, $\times$, and $*$.
Hence $R^k_{i,j}$ is regular.

With the proof of propositions 1 and 2 we have proved the theorem.

Now we will use this theorem to prove the main theorem.

Let a be the number of the initial state and b be the number of a final state.
$R^n_{a,b}$ is the set of all strings accepted by M in final state b.
It follows from the theorem just proved that $R^n_{a,b}$ is regular.

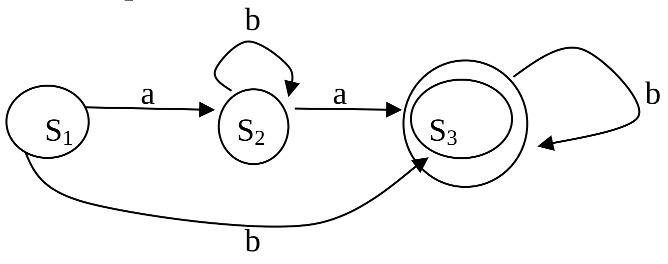
Let a be the number of the initial state and $b_1,...,b_m$ be the numbers corresponding to all the final states in M.
Then the language accepted by M is:

$$L(M) = R^n_{a,b1} \cup ... \cup R^n_{a,bm}$$

Since we have just seen that all the sets in this union are regular, L(M) is a union of regular sets, and hence L(M) is itself regular.
This proves the main theorem: every language accepted by a finite state automaton is regular.

We have now proved that all the language classes discussed here, right linear languages, left linear languages, finite state languages form one an the same class of languages, the class of regular languages.

**Example:**



| | |
|---|---|
| $R^3_{13} = R^2_{13} \cup ( R^2_{13} \times (R^2_{33})^* \times R^2_{33})$ | $R^0_{13} = \{b\}$ |
| $R^2_{13} = R^1_{13} \cup ( R^1_{12} \times (R^1_{22})^* \times R^1_{23})$ | $R^0_{11} = \{e\}$ |
| $R^2_{33} = R^1_{33} \cup ( R^1_{32} \times (R^1_{22})^* \times R^1_{23})$ | $R^0_{12} = \{a\}$ |
| $R^1_{13} = R^0_{13} \cup ( R^0_{11} \times (R^0_{11})^* \times R^0_{13})$ | $R^0_{22} = \{e, b\}$ |
| $R^1_{12} = R^0_{12} \cup ( R^0_{11} \times (R^0_{11})^* \times R^0_{12})$ | $R^0_{21} = Ø$ |
| $R^1_{22} = R^0_{22} \cup ( R^0_{21} \times (R^0_{11})^* \times R^0_{12})$ | $R^0_{23} = \{a\}$ |
| $R^1_{23} = R^0_{23} \cup ( R^0_{21} \times (R^0_{11})^* \times R^0_{13})$ | $R^0_{33} = \{e,b\}$ |
| $R^1_{33} = R^0_{33} \cup ( R^0_{31} \times (R^0_{11})^* \times R^0_{13})$ | $R^0_{31} = Ø$ |
| $R^1_{32} = R^0_{32} \cup ( R^0_{31} \times (R^0_{11})^* \times R^0_{12})$ | $R^0_{32} = Ø$ |

Hence:

| | |
|---|---|
| $R^1_{32} = Ø \cup (Ø \times \{e\} \times \{a\})$ | $R^1_{32} = Ø$ |
| $R^1_{33} = \{e,b\} \cup ( Ø \times \{e\} \times \{b\})$ | $R^1_{33} = \{e,b\}$ |
| $R^1_{23} = \{a\} \cup ( Ø \times \{e\} \times \{b\})$ | $R^1_{23} = \{a\}$ |
| $R^1_{22} = \{e,b\} \cup ( Ø \times \{e\} \times \{a\})$ | $R^1_{22} = \{e,b\}$ |
| $R^1_{12} = \{a\} \cup ( \{e\} \times \{e\} \times \{a\})$ | $R^1_{12} = \{a\}$ |
| $R^1_{13} = \{b\} \cup ( \{e\} \times \{e\} \times \{b\}$ | $R^1_{13} = \{b\}$ |

And:

$R^2_{33} = \{e,b\} \cup (Ø \times (\{e,b\}^* \times \{a\})$ $\qquad$ $R^2_{33} = \{e,b\}$

$R^2_{13} = \{b\} \cup ( \{a\} \times \{b\}^* \times \{a\})$

$R^3_{13} = \{b\} \cup ( \{a\} \times \{b\}^* \times \{a\}) \cup$
$\quad (\{b\} \cup ( \{a\} \times \{b\}^* \times \{a\})) \times \{e,b\}^* \times \{e,b\}$

Since $\{e,b\}^* \times \{e,b\} = \{e,b\}^*$ we simplify to:

$R^3_{13} = \{b\} \cup ( \{a\} \times \{b\}^* \times \{a\}) \cup$
$\quad (\{b\} \cup ( \{a\} \times \{b\}^* \times \{a\})) \times \{e,b\}^*$

Since $\{b\} \cup ( \{a\} \times \{b\}^* \times \{a\})$ falls under $(\{b\} \cup ( \{a\} \times \{b\}^* \times \{a\})) \times \{e,b\}^*$ we simplify to:

$R^3_{13} = (\{b\} \cup ( \{a\} \times \{b\}^* \times \{a\})) \times \{e,b\}^*$

This language is: $b^n \cup ab^mab^k$ $(n>0, \ m\geq 0, k\geq 0)$

Now the theorem promised above:

**Theorem:** For every non-determininstic finite state automaton there is an equivalent deterministic finite state automaton.

We start with some notation for non-deterministic finite state automata:

**$\delta[S,a]$: the set of states that you get to from S by a:**

$\boldsymbol{\delta[S,a]} = \{S_1\text{: } \langle S,a,S_1\rangle \in \delta\}$

This is the set of states that δ maps S and a onto.

**$\delta[S,\alpha]$: the set of states you get to from S by α:**

Let $\alpha = a_1 \dots a_n$

$\delta^1[S,\alpha] = \delta[S,a_1]$

$\delta^2[S,\alpha] = \{S_2\text{: } \exists S_1 \in \delta^1[S,\alpha]\text{: } \delta[S_1,a_2]=S_2\}$

…

$\delta^i[S,\alpha] = \{S_i\text{: } \exists S_{i-1} \in \delta^{i-1}[S,\alpha]\text{: } \delta[S_{i-1},a_2]=S_i\}$

$\boldsymbol{\delta[S,\alpha]} = \delta^n[S,\alpha]$

This is the set of states for which there is a derivation of α from S.

**Proof:**

Let M be a non-deterministic finite state automaton.

We define a deterministic finite state automaton K:

$S_K = \text{pow}(S_M)$

$F_K = \{S_K\text{: } \exists S_M\text{: } S_M \in S_K\}$ The set of sets of K-states that contain at least one final state of M. $S_{0,K} = \{S_{0,M}\}$

$\delta[\{S_1,\dots,S_i\},a] = \delta[S_1,a] \cup \dots \cup \delta[S_i,a]$

**Claim:** $\delta[\{S_0\},\alpha] = \delta[S_0,\alpha]$

Step 1: $\delta[\{S_0\},e] = \delta[S_0,e]$ (which is $\{S_0\}$ or Ø)

Step 2: Assume that $\delta[\{S_0\},\alpha] = \delta[S_0,\alpha]$

Then $\delta[\{S_0\},\alpha a] = \{S\text{: } \exists S_1 \in \delta[\{S_0\},\alpha]\text{: } S \in \delta(S_1,a)\}$

$= \{S\text{: } \exists S_1 \in \delta[S_0,\alpha]\text{: } S \in \delta(S_1,a)\}$ (by induction)

$= \delta[S_0,\alpha a]$

With this it follows that: $\delta[\{S_0\},\alpha] \in F_K$ iff $\delta[S_0,\alpha] \in F_K$

By definition of $F_K$: $\delta[S_0,\alpha] \in F_K$ iff $\exists S \in \delta[S_0,\alpha]\text{: } S \in F_M$

Hence: $\delta[\{S_0\},\alpha] \in F_K$ iff $\exists S \in \delta[S_0,\alpha]\text{: } S \in F_M$

This means that K generates α iff M generates α, hence K and M are equivalent.

## THE PUMPING LEMMA FOR REGULAR LANGUAGES.

Let A be a regular language. A is accepted by a finite state automaton M, with, say, n states.
Let $\alpha \in A$, $\alpha = a_1...a_m$, $m \geq n$.
Assume that there is a path through M for $a_1...a_m$ from $S_0$ to final state F.
Let's call the occurrences of states on that path $S^0...S^m$.
Since $m \geq n$, it is **not** possible that all the states $S^0...S^m$ are distinct, because $S^0...S^m$ form at least n+1 occurrences of states, and there are only n states.
This means that for some $j,k \leq n$: $S^j = S^k$ (let's assume $j < k$). In other words, $S^0...S^m$ contains a **loop**.

Suppose substring $a_{j+1}...a_k$ is a part of $a_1...a_m$ accepted by going through this loop once. We know then that: $1 \leq |a_{j+1}...a_k| \leq n$.
Now, instead of going through the loop from $S^j$ to $S^k$ in $S^0...S^m$, and then on to $S^{k+1}$, **we could have skipped the loop** and gone on directly from $S^j$ to $S^{k+1}$, and **the resulting string, α with** $a_{j+1}...a_k$ **replaced by e, would also have been accepted**.
Hence, α with $a_{j+1}...a_k$ replaced by e, is also in language A.
Similarly, **we could have gone through the loop twice,** and then go on as before, and the resulting string, α with $a_{j+1}...a_k$ replaced by $a_{j+1}...a_k a_{j+1}...a_k$, would also have been accepted, hence, α with $a_{j+1}...a_k$ replaced by $a_{j+1}...a_k a_{j+1}...a_k$ is also in language A.

Thus, if $a_1....a_j \mathbf{a_{j+1}...a_k} a_{k+1}...a_m \in A$, then

$$a_1....aj(\mathbf{aj_{+1}...a_k})^z ak_{+1}...a_m \in A, \text{ for every } z \geq 0.$$

Hence, for every sufficiently long strong string $a_1...a_m \in A$, we can find a substring that can be 'pumped' through the loop, and the result is also in A. This is the pumping lemma.

> **Pumping lemma for regular languages**:
> Let A be a regular language. There is a number n called the **pumping constant for** A(not greater than the number of states in the smallest automaton accepting A) such that:
> For every string $\varphi \in A$ with $|\varphi| \geq n$:
> $\varphi$ can be written as the concatenation of three substrings: $\alpha\beta\gamma$ such that:
> 1. $|\alpha\beta| \leq n$
> 2. $|\beta|>0$
> 3. for every $i \geq 0$: $\alpha\beta^i\gamma \in A$.

**Application:** $a^m b^m$ is not a regular language.

**Proof**:
Assume $a^m b^m$ is a regular language. Let n be the pumping constant for this language. Choose a number k such that 2k>n,and consider the string $a^k b^k \in a^m b^m$ of length 2k:

$$\underbrace{a\ldots\ldots\ldots\ldots\ldots a}_{k}\underbrace{b\ldots\ldots\ldots\ldots\ldots b}_{k}$$

According to the pumping lemma, we can write this string as $\alpha\beta\gamma$, where $\beta\neq e$,
$|\alpha\beta|\leq n$ and $\alpha\beta^i\gamma \in a^m b^m$.
Try to divide this string.
-If β consists only of a's, then pumping β will make the number of a's and b's not the same, hence the result is not in $a^m b^m$.
-If β consists only of b's, the same.
-If β consists of a's and b's, it is of the form $a^u b^z$.
So our string is:

$a\ldots\ldots\ldots\ldots\ldots(a^u b^z)\ldots\ldots\ldots\ldots\ldots b$

But then pumping β once gives:

$a\ldots\ldots\ldots\ldots\ldots(a^u b^z)\ (a^u b^z)\ldots\ldots\ldots\ldots\ldots b$

and this string has the a's and b's mixed, in the middle, hence it is not in $a^m b^m$.
Since these are the only three possibilities, we cannot divide this string in a way that satisfies the pumping lemma. This means that $a^m b^m$ does not satisfy the pumping lemma for regular languages, and hence $a^m b^m$ is not a regular language.

We will see shortly that it follows from this that English is not a regular language.

Note that the pumping lemma goes one way: if a language is regular, it satisfies the pumping lemma. But languages that satisify the pumping lemma are not necessarily regular. Let $L_1$ and $L_2$ be languages such that $L_1 \cap L_2 = \emptyset$. Assume that $L_1$ is regular, but $L_2$ is not, say, $L_2$ is intractable. Let n be the pumping constant for $L_1$ and let $L_1^n$ be the set of $L_1$ strings of length larger than n. Look at $L_1^n \times L_2$. Obviously, $L_1^n \times L_2$ is intractable in the same way that $L_2$ is. But, $L_1^n \times L_2$ satisfies the pumpinglemma, because the strings in $L_1^n$ do, by the fact that $L_1$ is regular.

## CLOSURE PROPERTIES OF REGULAR LANGUAGES.

We already know that if A,B are regular, then so are A*, $A \cup B$, and $A \times B$.

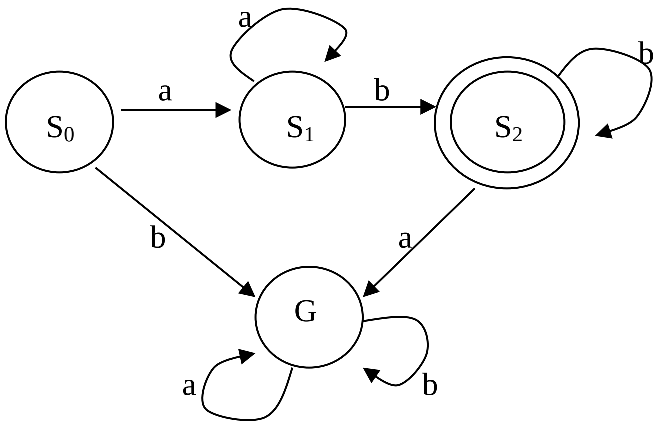
**Theorem:** Let L be a language in alphabet A ($L \subseteq A^*$).
If L is regular, then $A^* - L$ is regular.

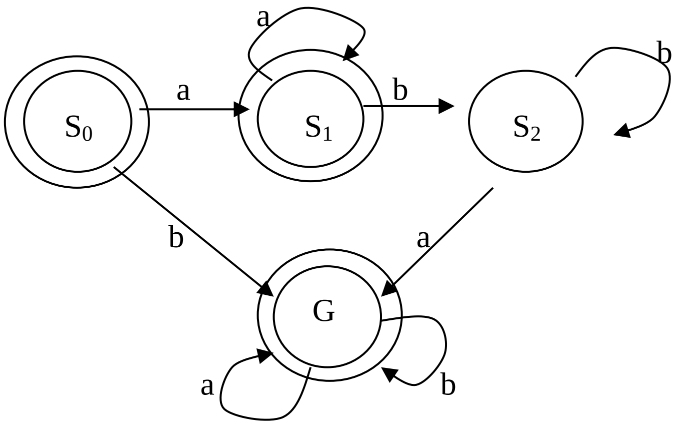
$A^* - L$ is the complement of L in A*. This means that the class of regular languages is closed under complementation.

**Proof**: Let L be regular, and let M be a **deterministic and total** automaton accepting L Make every final state in M non-final and every non-final state final.
The resulting automaton accepts $A^* - L$

Example:

$a^n b^m$ (n,m > 0)





Any string, as long as it is not just a's followed by b's:
Example: aaaabbbbbaaaab

**Corrollary**: If A and B are regular languages then $A \cap B$ is a regular language.

**Proof**:
Let A and B be regular languages in alphabet $\Sigma$ (which can be taken to be just the union of the symbols occurring in A and the symbols occurring in B).
$A \cap B = \Sigma^*-((\Sigma^*-A) \cup (\Sigma^*-B))$. That is, the operation of intersection can be defined as a sequence of compositions of the operations of complementation and union, Since we have proved the latter operations to be regular, and since sequences of compositions of regular operations are regular, intersection is regular.

Making an intersection automaton is a lot of work , though.
-Start with a deterministic automaton of A and a deterministic automaton for B.
-Take for both of them the complement automaton (i.e. switch final and non-final states).
-For the resulting two automata, $M_1$ and $M_2$ form the union automaton. This goes in the same way as we did for right linear grammars:
make the states of the two automata disjoint, add a new initial state, add for every arrow leaving $M_1$'s old initial state to some state $S_i$ a similar arrow from the new initial state to $S_i$, and the same for any such arrow leaving $M_2$'s old initial state.
Make the new initial state a final state if one of the old initial states was final.
-Next convert this automaton to a deterministic automaton (since the union procedure tends to give you a non-deterministic automaton). And finally take the complement automaton of the result. This will be an automaton for the intersection.
I will give a simple construction of an intersection automaton later in this course.

These resuls mean that the set of regular languages in a certain alphabet form a Boolean algebra.

Let A and B be alphabets.
A **homomorphism** from A* into B* is a function that maps strings in A* onto strings in B* in which the value for a complex string in A* is **completely determined** by the values for the **symbols** of A. Formally:

A **homomorphism** from A into B is a function $h:A^*\rightarrow B^*$ such that:

1. $h(e)=e$
2. for every string in A* of the form $\alpha a$, with $\alpha \in A^*$ and $a \in A$:
   $h(\alpha a) = h(\alpha)^\wedge h(a)$.

So: if h(b)=bb and h(a)=aa, then:
h(bba)= h(bb)h(a) = h(bb)aa = h(b)h(b)aa = h(b)bbaa = bbbbaa.

Let L be a language in alphabet A, and let $h:A^*\rightarrow B^*$ be a homomorphism, then: the **homomorphic image of L under h**, h(L) is given by:
$h(L) = \{h(\alpha): \alpha \in L\}$

**Theorem:** If L is a regular language in alphabet A and $h:A^*\rightarrow B^*$ is a homomorphism, then h(L) is a regular language in alphabet B.
**Proof:** below

**Example:**
Let A = {a,b}.  Let B = {a,b,c}
Let $h:A^* \rightarrow B^*$ be a homomorphism such that:

$h(e)=e$
$h(a)=aa$
$h(b)=cc$

We know that: $a^n b^m$ (n,m>0) is a regular language.
$h(a^n b^m\ (n,m>0)) = (aa)^n(cc)^m\ (n,m>0)$.
It follows that: $(aa)^n(cc)^m$ (n,m>0) is also a regular language.

**Example:**
Let A = {a,b,c}.
Let $h:A^* \rightarrow A^*$ be a homomorphism such that:

$h(e)=e$
$h(a)=a$
$h(b)=b$
$h(c)=e$

We have proved that $a^n b^n (n \geq 0)$ is **not** a regular language.
We look at: $a^n c b^n (n \geq 0)$.
$h(a^n c b^n (n \geq 0)) = a^n b^n (n \geq 0)$.
Consequently we know: if $a^n c b^n (n \geq 0)$ were regular, $a^n b^n (n \geq 0)$ would be regular. But $a^n b^n (n \geq 0)$ is not regular. Hence:
$a^n c b^n (n \geq 0)$ is **not** regular.

Let $h:A^* \rightarrow B^*$ be a homomorphism.
For each string $\beta \in B^*$, we define:
$h^{-1}(\beta) = \{\alpha \in A^*: h(\alpha)=\beta\}$
We call $h^{-1}$ the **inverse homomorphism** of h.
(Note: $h^{-1}$ is not a function from B* into A*, but from B* into pow(A*).)

For $L \subseteq B^*$ we define:
$h^{-1}(L) = \{\alpha \in A^*: h(\alpha) \in L\}$

**Theorem**: If L is a regular language in alphabet B and $h:A^* \rightarrow B^*$ is a homomorphism, then $h^{-1}(L)$ is a regular language in alphabet A.
**Proof:**
Let $h: A^* \rightarrow B^*$ is a homomorphism and L a regular language in B*.
Let M be a deterministic total finite state automaton for L.

We simplify the notation introduced for Non-deterministic finite state automata:
Let $\beta = b_1 \dots b_n \in B^*$

$\delta^1[S,\beta] = \delta[S,b_1]$
$\delta^2[S,\beta] = \delta[\delta^1[S,\beta],b_2]$
…
$\delta^i[S,\beta] = \delta[\delta^{i-1}[S,\beta],b_i]$

$\delta[S,\beta] = \delta^n[S,\beta]$

We turn M into M': a deterministic total finite state automaton on A* by defining:

$\delta[S,a] = \delta[S,h(a)]$

**Claim:** M' acceps $\alpha$ iff M accepts $h(\alpha)$

Case 1: M' accepts a iff M accepts h(a)
M' gets to the same state final or non-final state after a where M gets after h(a), by the construction.

Case 1: Assume M' accepts $\alpha$ iff M accepts $h(\alpha)$
Then M' accepts $\alpha a$ iff M accepts $h(\alpha)h(a)$.
Again, this is obvious from the construction.

Let A and B be alphabets.
A **substitution from** A into B is a function that maps every string of A* onto a **set** of strings in B which is determined by the values of the symbols in the following way: The strings in the set associated with a complex string $\alpha$ are gotten by substituting all values for the symbols in $\alpha$ at the place where they occur. Formally:

A **substitution** from A into B is a function $s{:}A^*\rightarrow pow(B^*)$ such that:
1. $s(e)=\{e\}$
2. For any string of the form $\alpha a$, with $\alpha \in A^*$ and $a \in A$:
   $s(\alpha a) = s(\alpha)\times s(a)$

If L is a language in alphabet A and s a substitution from A into B, then, **the substitution language of L relative to s**, s(L), is given by:
$s(L) = \cup_{\alpha\in L}s(\alpha)$.

Idea:
$s(a) = \{\alpha_1, \alpha_2\}$
$s(b) = \{\beta\}$
$s(c) = \{\gamma_1, \gamma_2\}$

$$\begin{tikzcd}
a \arrow[d] & \wedge \arrow[d] & b \arrow[d] & \wedge \arrow[d] & c \arrow[d] & \in L \\
\{\alpha_1, \alpha_2\} & \times & \{\beta\} & \times & \{\gamma_1, \gamma_2\} & \subseteq s(L)
\end{tikzcd}$$

**Example**:
$ab^ncd^m$ (n,m>0) is a regular language in A={a,b,c,d}
(a, followed by as many b's as you want, followed by c, followed by as many d's as you want).

We take alphabet B = {John, Bill, Mary, and, walk, talk, sing}
and we take s, a substitution from A into B given by:

s(a) = {John, Bill, Mary}
s(b) = {and John, and Bill, and Mary}
s(c) = {walk, talk, sing}
s(d) = {and walk, and talk, and sing}

s(abbcddd) =

{John, Bill, Mary} × {and John, and Bill, and Mary} × {and John, and Bill, and Mary} × {walk, talk, sing} × {and walk, and talk, and sing} × {and walk, and talk, and sing} × {and walk, and talk, and sing}.

So, one of the strings in the substitution language of abbcddd is:

John and Bill and Mary walk and talk and sing.

Another one is:

Mary and Mary and Mary talk and talk and talk.

s($ab^ncd^m$ (n,m>0)) =
{John, Bill, Mary} × $\{$and John, and Bill, and Mary$\}^+$ × {walk, talk, sing} × $\{$and walk, and talk, and sing$\}^+$.

which contains any string, starting with either *John* or with *Bill* or with *Mary*, followed by one or more occurrences of strings in {*and John*, *and Bill*, *and Mary*}, followed by one of the items *walk* or *talk* or *sing*, ending with one or more of the items in {*and walk*, *and talk*, *and sing*}.

**Theorem:** If L is a regular language in alphabet A and s is a substitution from A* into pow(B*), that maps every symbol in A onto a regular subset of B*, then s(L) is a regular language in alphabet B.

**Proof:**
Let L be a regular language in A.
This means that L is derived with ∪, ×, * from finite languages $L_1,\dots,.L_n$

1. If $a \in A$ then s(a) is a regular language.
   If $\alpha \in A^*$, $\alpha = a_1\dots a_n$, then $s(\alpha) = s(a_1)\times\dots\times s(a_n)$. Since for each $i \leq n$: $s(a_i)$ is regular, α is regular.

   If L is finite, $L = \{\alpha_1,\dots,\alpha_n\}$.
   $s(L) = s(\{\alpha_1,\dots\alpha_n\}) = s(\{\alpha_1\}) \cup ... \cup s(\{\alpha_n\})$.
   Since for each $i \leq n$: $s(\{\alpha_i\})$ is regular, $s(\{\alpha_1\}) \cup ... \cup s(\{\alpha_n\})$ is regular.
   Hence s(L) is regular.
2. Assume $L = L_1 \cup L_2$, with $L_1$ and $L_2$, $s(L_1)$ and $s(L_2)$ regular.
   $s(L_1 \cup L_2) = s(L_1) \cup s(L_2)$, hence $s(L_1 \cup L_2)$ is also regular, so s(L) is regular.
3. Assume $L = L_1 \times L_2$, with $L_1$ and $L_2$, $s(L_1)$ and $s(L_2)$ regular.
   $s(L_1 \times L_2) = s(L_1) \times s(L_2)$, hence $s(L_1 \times L_2)$ is also regular, so s(L) is regular.
4. Assume $L = L_1^*$, with $L_1$ and $s(L_1)$ is regular.
   $s(L_1^*) = s(L_1)^*$, hence $s(L_1^*)$ is also regular, so s(L) is regular.

**Corrollary:**
If L is a regular language in alphabet A and h is a homomorphism from A* into B* then s(L) is a regular language.

**Proof:**
Instead of $h(\alpha)=\beta$, we can write $h(\alpha)=\{\beta\}$, without loosing any information. This means that homomorphisms are a special case of substitutions, with the values singleton sets, hence finite, hence regular. This means that the above theorem applies to them.

With the theorem, we don't have to prove separately that this language is a regular language, that follows from the theorem.

We see that, with the notions of homomorphism, inverse homomorphism, substitutions, we can extend our theorems from languages that look like toy languages (with little a's and b's) to languages that look suspiciously like natural languages.

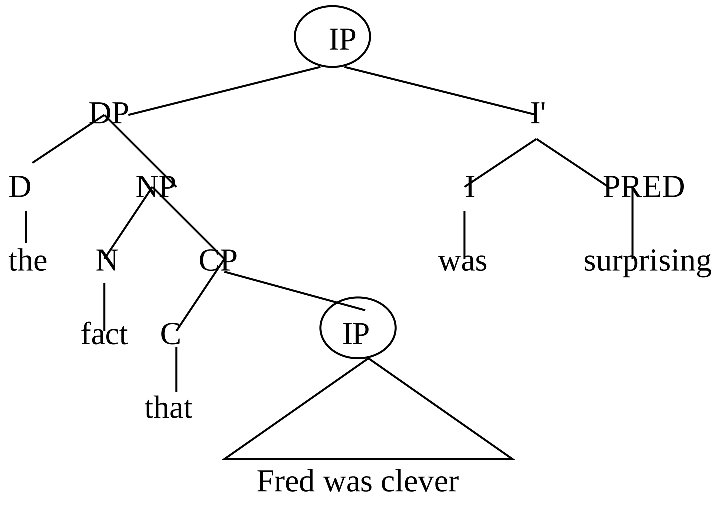
We use the fact that the intersection of regular languages is regular and the fact that regular languages are closed under homomorphisms to prove that the natural language English is not a regular language:

**Theorem**: English is not a regular language.

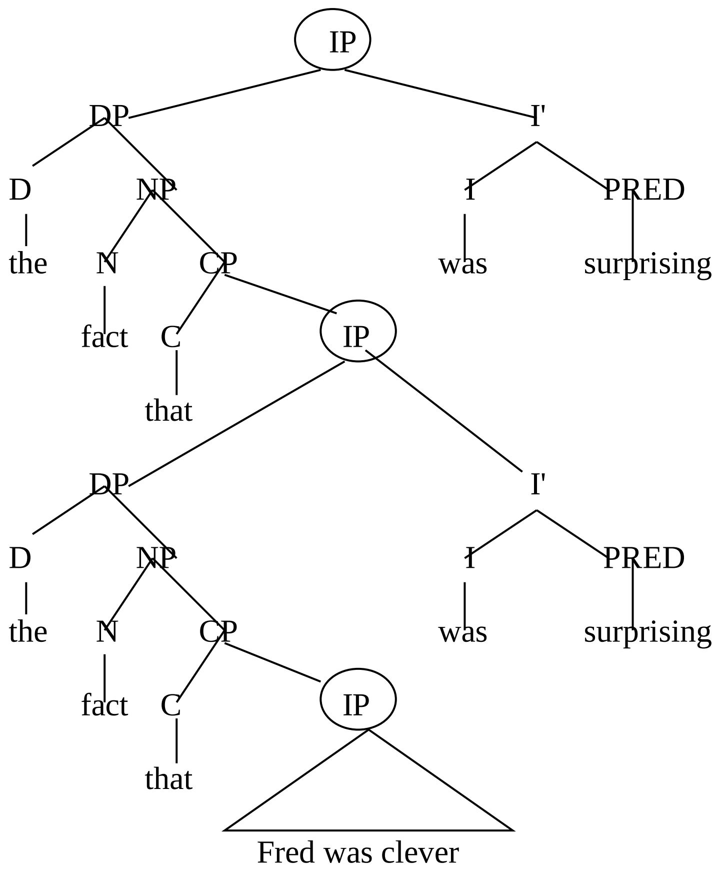
**Proof**:
Let the set of grammatical strings of English be E.
I specify a sequence of strings $\alpha_1,\alpha_2$,....

$\alpha_1$ = *the fact that* Fred was clever **was surprising**
$\alpha_2$ = *the fact that the fact that* Fred was clever **was surprising was surprising**
$\alpha_3$= *the fact that the fact that the fact that* Fred was clever **was surprising was surprising was surprising**
....
and we set: L = $\{\alpha_1,\alpha_2,\alpha_3,...\}$

We are dealing with the following structure:



The idea is, of course, that we expand the tree by looping the bit between the IP nodes:



And L is:

L = *the fact that*$^n$ Fred is clever **was surprising**$^{\mathbf{n}}$**,** (n>0)

Choose the following homomorphism:
h(*the fact that*)=a, h(Fred is clever)=c, h(**was surprising**)=b**..**
Then the homomorphic image of L is $a^n c b^n$ which we showed to be not regular.
Consequently: **L is not a regular language**.

Now we look at a bit wider language, L':

L' = *the fact that*$^n$ Fred is clever **was surprising**$^{\mathbf{m}}$**,** (n,m>0)

We chose a homomorphism such that:
h(a) =*the fact that*, h(c)=Fred is clever, h(b)=**was surprising.**
L' is the homomorphic image of $a^n c a^m$(n,m>0), which we showed earlier to be regular.
Hence: **L' is regular**.

Now we observe the **empirical fact about English**:

**Empirical Fact**: $E \cap L' = L$

The only strings **of L'** that are grammatical in English are the strings sentences in L.

So we have three facts:

1. L is not regular.
2. L' is regular.
3. $L = L' \cap E$.

Suppose English were a regular language. Then both E and L' would be regular.
The intersection of two regular languages is regular, hence L would be regular.
But L is not regular. Since L' is regular, it follows that E is not regular.
This completes the proof.

Other example:

The man who $\alpha^n$ sincerely $\beta^n$ believes that he is crazy, is indeed crazy $n \geq 0$
$\alpha$ = the man who
$\beta$ = believes that he is crazy