26th September 2000

1. Strings and Languages

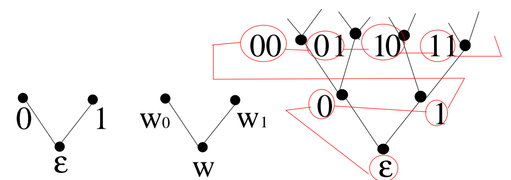
An **alphabet** Σ is just a *finite* set of symbols, e.g. $\Sigma = \{1\}$ (a *tally*), $\Sigma = \{0,1\}$ (*bits*), $\Sigma = \{a,...,z,A,...,Z,!,?,...\}$, Σ = {all words in the OED}. In the final example, each word is a *single* symbol. A string (over Σ) is a **finite** sequence of symbols from Σ. For example, if $\Sigma = \{0,1\}$, then examples of strings are 011, 1, 0, 00000, etc. If Σ = {all words in the OED}, then an *example* of a string could be "to be or not to be" (individual symbols underlined for *clarity*).

The string with **no** symbols is called the empty string, ε. The collection of **all** strings over an alphabet Σ is denoted by Σ^* ("sigma star"). If $x \in \Sigma^*$, then $|x|$ is the *length* of x, e.g. $|1001| = 4$; $|\varepsilon| = 0$. If $a \in \Sigma$, then $|x|_a$ is the number of times the *symbol* a appears in x. Example: $|0110|_0 = 2$. If $a \in \Sigma$, then $a^0 = \varepsilon$ (by *definition*), and $a^n = aaaa...aa$ (n times, with $n \geq 1$). For example, $0^3 1^2 0^2 = 0001100$.

Any subset L of Σ^* is called a language. **Examples**: (1) Let Σ = {all words in the OED}. Then Σ^* = all possible *sequences* of words. Define $L \subseteq \Sigma^*$ to be all sequences of English words which make *grammatical sense*. It follows that L = "The English language". (2) Let $\Sigma = \{0,1\}$. Then Σ^* consists of all possible binary strings. Define $L \subseteq \Sigma^*$ to be all (*syntactically*) correct programs in Java converted into binary.

Let $x, y \in \Sigma^*$. The string xy is called the *concatenation* of x and y (in that order). For example, if x = 00, and if y = 11, then xy = 0011, and yx = 1100. **Properties**: $(xy)z = x(yz)$; $\varepsilon x = x = x\varepsilon$; define $x^0 = \varepsilon$; and $x^n = xxx..x$ (n times, with $n \geq 1$). If x = uvw, where u, v and w are strings, then u is the *prefix*, v is the *factor*, and w is the *suffix*.

Ordering the elements of Σ^*. Let $\Sigma = \{0,1\}$, and assume that $0 < 1$. *List* the elements of Σ^* as shown on the right. Note that $w \in \{0,1\}^*$, and that we have *right tree ordering*, which gives the list ε, 0, 1, 00, 01, 10, 11, 000, ...



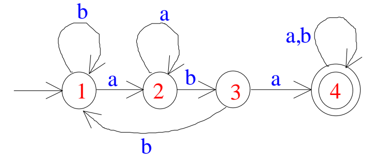
Language Operations

Let L and M be **languages** over Σ. Language Operations include $L \cap M$; $L \cup M$; $\bar{L}$ (the *complement* of L in Σ^*: $\bar{L} = \Sigma^* \backslash L$, all *strings* in Σ^* not in L); and $LM = \{xy: x \in L, y \in M\}$, the *product* of L and M. For example, if $L = \{a, ba, b^2\}$, and if $M = \{a^2, ab^2, ba\}$, then $LM = \{aa^2, aab^2, aba, baa^2, baab^2, baba, b^2a^2, b^2ab^2, b^2ba\}$.

28th September 2000

Now $L^0 = \{\varepsilon\}$ by definition; $L^n = L...L$ (n times) ($n \geq 1$); and $L^* = L^0 \cup L^1 \cup L^2 \cup ... = \cup_{L=0}^{\infty} L^i$. For example, if $L = \{a,b\}$, then $L^* = \{a,b\}^*$ in the usual sense. If $L = \{ab, ba\}$, then $L^* = \{\varepsilon, ab, ba, (ab)(ab), (ba)(ba), ..., (ab)(ab)(ab), ...\}$. This is the **Kleene star** of L. The **aim** of this course is to prove Kleene's Theorem — the characterisation of *recognisable* languages.

Deterministic Finite State Automata



Circles represent *states*, and these can be used to store information — memory. A **single arrow** pointing to a circle points to the *initial state*. **Double** ringed circles represent *terminal* or *accepting* states. $\Sigma = \{a,b\}$ is an alphabet. Arrows labelled by *elements* of Σ are transitions. The aim of this machine is to **recognise** a language $L \subseteq \Sigma^*$.

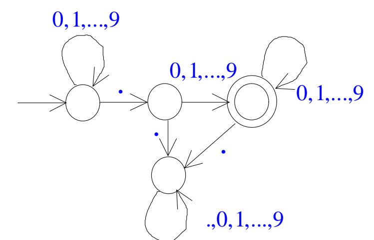
Example: ↘aaba in *state* 1; then a↘aba in *state* 2; aa↘ba in 2; aab↘a in 3; and aaba↘ in 4. So aaba is **accepted** by this machine (because state 4 is an *accepting/terminal* state). Note that the (↘) arrows are meant to point to the **top** of the *next* letter. Now *try* ↘bab (1); b↘ab (1); ba↘b (2); and bab↘ (3). **Not** accepted.

If A is this machine, then L(A) is the set of all *strings* which lead from the **initial** state to a **terminal** state. It follows that L(A) is the language *accepted* by A. This machine is said to be deterministic because the current state and the input letter determines the next state (each *circle* has two arrows coming out of it).

Definition: *A deterministic finite state automaton is represented by* $\underline{A} = (S, \Sigma, s_0, \delta, F)$, where S = a *finite* set of states; Σ = the input *alphabet*; s_0 = the *initial* state; F = the *terminal* or *accepting* states; and $\delta: S\times\Sigma \to S$ *is the transition* function, where $\delta(s,a)$ (s is the *current* state, a is the *input* letter, and the whole thing is the *next* state) is often written as $\delta(s,a) = s\bullet a$. In figure 1 (the *transition diagram above*), $S = \{1,2,3,4\}$, $\Sigma = \{a,b\}$, $s_0 = 1$, and $F = \{4\}$. We can represent δ by a *transition table* as shown which, with the **extra** symbols, can represent all the information.

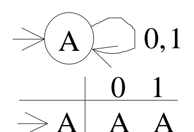
		a	b
→	1	2	1
	2	2	3
	3	4	1
◎	4	4	4

How to *process strings*. Consider that $a_1 \ldots a_n$ is our **input** string. To *process* the string, we **manipulate** the following expression: $(\ldots(((s_0\bullet a_1)\bullet a_2)\bullet a_3)\ldots)\bullet a_n$. The function which *processes* strings is $\delta^*: S\times\Sigma^* \to S$, the *extended state transition function*: $\delta^*(s,\varepsilon) = s$ $(s \in S)$; $\delta^*(s,a) = \delta(s,a)$ $(s \in S,\ a \in \Sigma)$; and $\delta^*(s,ax) = \delta^*(\delta(s,a), x)$ $(a \in \Sigma,\ x \in \Sigma^*,\ s \in S)$. **Definition**: $L(\underline{A}) = \{x \in \Sigma^*: \delta^*(s_0,x) \in F\}$ is the language *accepted/recognised* by A. As **before**, $\delta^*(s,x) = s\bullet x$.



Examples: (i) *Compiler*. This converts a high level language such as C++ into **machine code**. The first stage is called *lexical analysis*. Tokens such as (,), {, }, begin, end, if, 2.314, etc. are recognised using f.s.a. For example, the diagram shown recognises numbers in *decimal form,* where $\Sigma = \{., 0, 1, \ldots, 9\}$. (ii) *Text Processing* e.g. searching for words in text. (iii) Important in *Combinatorial Group Theory.*

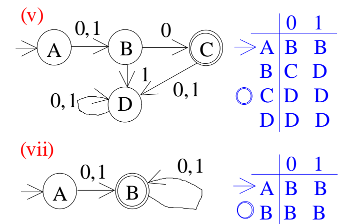
Exercises



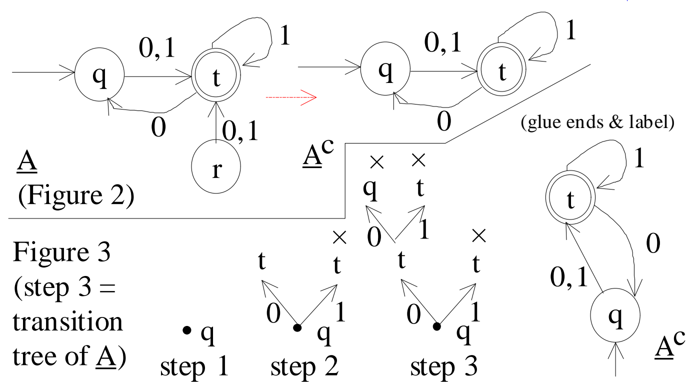
	0	1
→ A	A	A

Q: Let $\Sigma = \{0,1\}$. **Construct** deterministic f.s.a.'s which recognise the *languages* (i) ϕ; (ii) $\{\varepsilon\}$; (iii) $\{0\}$; (iv) $\{0101\}$; (v) $\{00,10\}$; (vi) Σ^*; and (vii) $\Sigma^*\backslash\{\varepsilon\}$. A: (i) $L(\underline{A}) = \phi$; $S = \{A\}$; $\Sigma = \{0,1\}$; $s_0 = A$; $F = \phi$; and the transition *table* and the *diagram* are as shown on the right.

(v) L(<u>A</u>) = {00, 10}; S = {A, B, C, D}; Σ = {0,1}; s_0 = A; F = C; and the transition *table* and the *diagram* are as shown on the right. (vii) L(<u>A</u>) = Σ*\{ε}; S = {A,B}; Σ = {0,1}; s_0 = A; F = B; and the transition *table* and the *diagram* are as shown on the right.



<u>Definition</u>: Let <u>A</u> be a f.s.a. We say that a *state* s ∈ S is **accessible** if ∃ x ∈ Σ* s.t. s_0•x = s. <u>A</u> is said to be *connected* if every state is accessible. Looking at figure 2 on the right, {q,t} are accessible, while {r} is **not** accessible. If <u>A</u> is not connected, *construct* a connected machine $\underline{A}^c$ s.t. L($\underline{A}^c$) = L(<u>A</u>). All you do is to remove all the *non accessible* states from <u>A</u> and their associated transitions.



There is an algorithm for constructing $\underline{A}^c$ from A based on constructing *"the transition tree"* of <u>A</u>. Figure 3, the transition tree, has as labels precisely the **accessible** states. To get $\underline{A}^c$, glue the leaves of the tree to the *interior* nodes with some label. **<u>Non-deterministic automata</u>**.

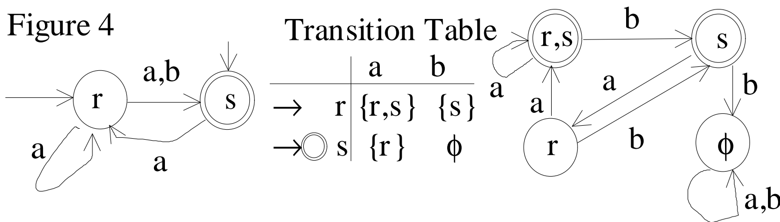


Figure 5 The *Determined* version of Fig. 4 using the subset convention

In *figure 4*, shown on the left, there are 2 **initial** states; more than one transition **labelled** 'a' comes out of a state; and there is a **missing** transition (no b from s). This is an example of a *non-deterministic automaton*, where non-deterministic means that the current state and the current input does <u>not</u> uniquely determine the next state.

<u>Definition</u>: <u>A</u> = (S, Σ, s_0, δ, F) is a <u>non-det. f.s.a.</u> if S = a finite number of states; Σ = the input alphabet; s_0 ⊆ S is the set of *initial* states; F ⊆ S is the set of *terminal* states; and δ: S×Σ→P(S) is the *transition function*. (P(S) = the power set of S). Therefore, δ(s,a) = {s_1,...s_n} (s = *current* state; a = *input* letter; {s_1,...,s_n} = set of next states), and there is the possibility that the situation δ(s,a) = ϕ is not excluded. (If this *is* the case, then we say that the machine *crashes*).

How are the strings *accepted* by such machines, e.g. abaa in figure 4? Here, the different **combinations** we can have are r—*a*→s—*b*→crash; r—*a*→r—*b*→s—*a*→r—*a*→((s)); r—*a*→r—*b*→s—*a*→r—*a*→r; s—*a*→r—*b*→s—*a*→r—*a*→r; and s—*a*→r—*b*→s—*a*→r—*a*→((s)). We say that abaa is *accepted* because it labels some path from one of the **initial** states to one of the **terminal** states.

<u>Extended State Transition Function</u>. δ*: S×Σ*→P(S) is *defined* by: δ*(s,ε) = {s}; δ*(s,a) = δ(s,a) (a ∈ Σ); and δ*(s,ax) = δ*(s_1,x)∪δ*(s_2,x)∪...∪δ*(s_n,x), where δ(s,a) = {s_1,...,s_n}. Q: In *figure 4*, calculate δ*(r,↘abaa). A: δ*(r,↘abaa) = δ*(r,↘baa)∪δ*(s,↘baa) = δ*(s,↘aa)∪ϕ = δ*(r,↘a) = δ(r,↘a) = {r,s}.

Definition: The language *accepted* by a non deterministic A is given by $L(\underline{A}) = \{x \in \Sigma^*: \delta^*(s,x) \cap F \neq \phi \text{ for some } s \in s_0\}$.

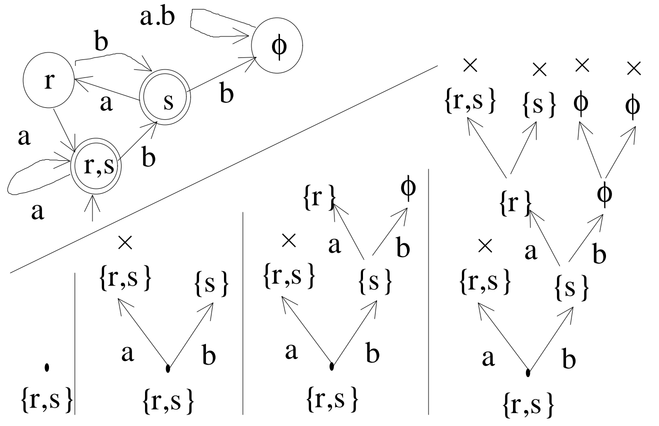
3rd October 2000

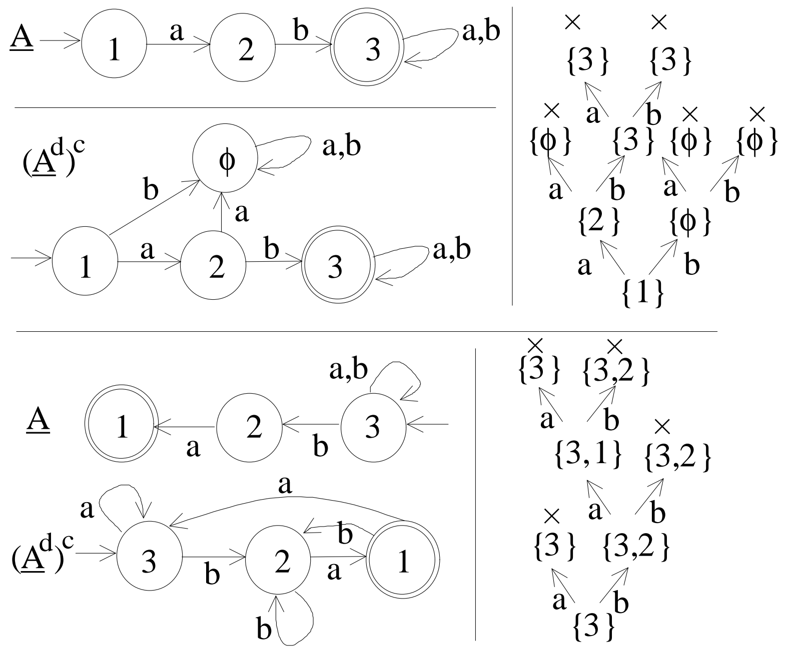
Challenge: convert a *non-det.* automaton A into a det. automaton $\underline{A}^d$ s.t. $L(\underline{A}) = L(\underline{A}^d)$. Method: **Subset construction**. Let $\underline{A} = (S, \Sigma, s_0, \delta, F)$ be non deterministic. Construct a *deterministic machine* called $\underline{A}^d$ from A as follows: the **states** of $\underline{A}^d$ are P(S); Σ is the alphabet; the *initial* state of $\underline{A}^d$ is s_0 ($\subseteq S$); the *terminal* states of $\underline{A}^d$ are $\{Q \subseteq S: Q \cap F \neq \phi\}$; and the transition function is given by $\delta^d: P(S) \times \Sigma \to P(S)$, where $\delta^d(Q,a) = \cup_{q \in Q} \delta(q,a)$, with $a \in \Sigma$.

Let us now *apply* this construction to figure 4. States of the *old* machine = {r,s}. States of the *new* machine = {(r,s), (s), (r), (ϕ)}. See figure 5 on the previous page for the picture. Example of calculation: {r,s}•a = r•a∪s•a (where r•a and s•a are in the **old** machine) = {r,s}∪{r} = {r,s}. *And* {r,s}•b = r•b∪s•b = {s}∪ϕ = {s}.

Theorem 1: Let $\underline{A} = (S, \Sigma, s_0, \delta, F)$ be a non-det. f.s.a. Then $\underline{A}^d$ is a det. f.s.a. s.t. $L(\underline{A}) = L(\underline{A}^d)$. *Sketch Proof*: Denote by $(\delta^d)^*$ the *extended state transition function* in $\underline{A}^d$. $x \in L(\underline{A}^d) \Leftrightarrow (\delta^d)^*(s_0, x) \in F^d$ (F^d = the set of *terminal states* in $\underline{A}^d$) $\Leftrightarrow (\delta^d)^*(s_0,x) \cap F \neq \phi$. **Let** $x = a_1 \ldots a_n$ for some $a_i \in \Sigma$. Then we *have* $s_0 \text{—} a_1 \to s_1 \text{—} a_2 \to s_2 \ldots \text{—} a_n \to s_n \Leftrightarrow a_1 \ldots a_n$ labels a *path in* A beginning at some element of s_0 and ending at an **element** of F $\Leftrightarrow x \in L(\underline{A})$.

The machine $\underline{A}^d$ has *many* states: if A has n states, then $\underline{A}^d$ has 2^n states. Sometimes, this is the *best we can do* — but often we can do better. **Idea:** (The Modified Subset Construction): Construct $(\underline{A}^d)^c$ directly from A *without* using $\underline{A}^d$. We do this by constructing the *transition tree* of $\underline{A}^d$ directly from A. Then *we construct* $(\underline{A}^d)^c$ by "*gluing*".

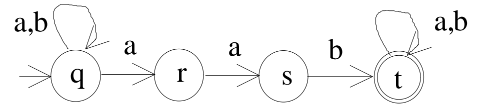




Example (the use of non-det. f.s.a.'s in **designing** machines): Show that the *language* {a b}{a,b}* is recognisable (the set of all strings which **begin** with ab). A: We can easily draw up a non-det. machine A as shown. We then use the *transition tree method* to get $(\underline{A}^d)^c$, and then *glue* things together to get the *final* diagram as shown.

Q: **Show** that {a,b}*{ba} is *recognisable*. A: As shown on the left.

Q: Consider the *non-det. f.s.a.* shown on the right. (i) Describe L(A); and (ii) construct $(\underline{A}^d)^c$ using the *transition tree* method. A: (i) L(A) = {a,b}*{aab}{a,b}*. (ii) Try this *yourself*. It is like the two examples on the **previous** page.



Q: If L(A) = $\{a^nb^n: n \geq 0\}$ = {ε, ab, aabb, aaabbb, ...}, construct a *f.s.a.* which recognises this language. A: If you tried to draw a diagram, you would end up with an infinite diagram. This is not a **finite** state automaton. It can be proved that this language is *not* recognisable. Tip for answering questions: take particular note in the way in which you *describe* the language a diagram recognises — do you use **words** such as "All strings in Σ* which contain ab", or a **list**?

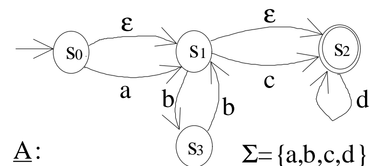
10th October 2000

Question: Given A, how do we find L(A)? We will develop an *algorithm* to solve this problem. Q: Given a language L, how do we find (if it exists) a machine A s.t. L(A) = L? Further, if L = L(A), and if M = L(B), how do we build a machine C such that L(C) = LM? Also, how do we build a machine D s.t. L(D) = L*? Idea: *det. f.s.a.* ® *non-det. f.s.a.* ® *non-det.* ***e**-machines*.

Non Deterministic Automata with ε-transitions

Definition: Such a machine A is defined as follows: A = (S, Σ, s_0, δ_ε, F). All symbols mean what they meant in the *non-det. case,* except that δ_ε: S×{Σ∪{ε}}→P(S). As *usual,* $\delta_\varepsilon(s,a)$ = s•a, where a ∈ Σ∪{ε}. At this stage, ε is just an *extra* letter of the alphabet. The question is, how is such a machine going to **process** strings?

There are **two** equivalent ways of answering this question: (1) ↘bb in s_0 → (by ε transition) ↘bb (S_1) → b↘b (S_3) → bb↘ (S_1) → (by ε-transition) bb↘ (S_2). Since S_2 is terminal, and since A has read bb this string is *accepted*. (2) We rely on the *following* facts: For x ∈ Σ*, εx = x = xε; and $\varepsilon^m = \varepsilon$ for any *natural* number m. Let a ∈ Σ, then $\varepsilon^m a \varepsilon^n$ = a for *any* m, with n ≥ 0. So bb = $\varepsilon^x b \varepsilon^y b \varepsilon^z$, and in *particular*, εbbε = bb. This labels a **path** in A starting at the initial state and finishing at a *terminal* state — so bb is accepted. **Example**: a = aε. This labels a "*good*" path in the machine, so that a is *accepted*. **Examples**: d = εεd (accepted); ε = εε (accepted). Let us now **formalise** these examples.

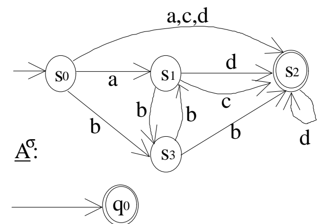


To determine whether a *string* x ∈ L(A), we need the *following* definition: Let x = $a_1...a_n$, with a_i ∈ Σ. Then an ***e**-extension* of x is any string of the **form** $\varepsilon^{p_1}a_1\varepsilon^{p_2}a_2...\varepsilon^{p_n}a_n\varepsilon^{p_{n+1}}$ for some *natural numbers* p_1, ..., p_{n+1}. **Definition**: x ∈ L(A) ⇔ some ***e**-extension* of x labels a path from an **initial** state to a **terminal** state.

Now for a *procedure* which will convert an ε-machine into a *non-det. machine*. **Idea**: $s\bullet(\varepsilon^m a \varepsilon^n) = ((s\bullet\varepsilon^m)\bullet a)\bullet\varepsilon^n$ (with a ∈ Σ). *Definition*: Let t ∈ S, then the ε-closure of t is Λ(t) = {all states in A which can be *reached* from t using only ε's}. *Example*: Λ(s_0) = {s_0,s_1,s_2}; Λ(s_1) = {s_1,s_2}; Λ(s_2) = {s_2}; and Λ(s_3) = {s_3}.

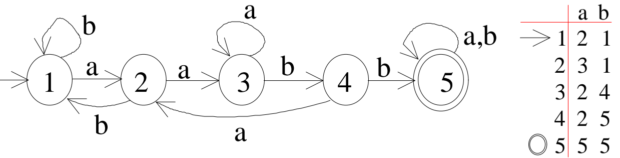
More *generally*, if $Q \subseteq S$, *define* $\Lambda(Q) = \cup_{q \in Q} \Lambda(q)$. Using this *notation*, a letter $a \in \Sigma$ is processed in the *following* way: $\Lambda(\Lambda(s)\bullet a) = \{s\bullet(\varepsilon^m a \varepsilon^n)\ m, n \geq 0\}$. We can now use the *bottom* table to construct a non-det. f.s.a. *without* ε-transitions which recognises the **same** language.

States	Λ(s)	Λ(s)•a	Λ(s)•b	Λ(s)•c	Λ(s)•d	Λ(Λ(s)•a)	Λ(Λ(s)•b)	Λ(Λ(s)•c)	Λ(Λ(s)•d)
s_0	$\{s_0,s_1,s_2\}$	$\{s_1\}$	$\{s_3\}$	$\{s_2\}$	$\{s_2\}$	$\{s_1,s_2\}$	$\{s_3\}$	$\{s_2\}$	$\{s_2\}$
s_1	$\{s_1,s_2\}$	ϕ	$\{s_3\}$	$\{s_2\}$	$\{s_2\}$	ϕ	$\{s_3\}$	$\{s_2\}$	$\{s_2\}$
s_2	$\{s_2\}$	ϕ	ϕ	ϕ	$\{s_2\}$	ϕ	ϕ	ϕ	$\{s_2\}$
s_3	$\{s_3\}$	ϕ	$\{s_1\}$	ϕ	ϕ	ϕ	$\{s_1,s_2\}$	ϕ	ϕ



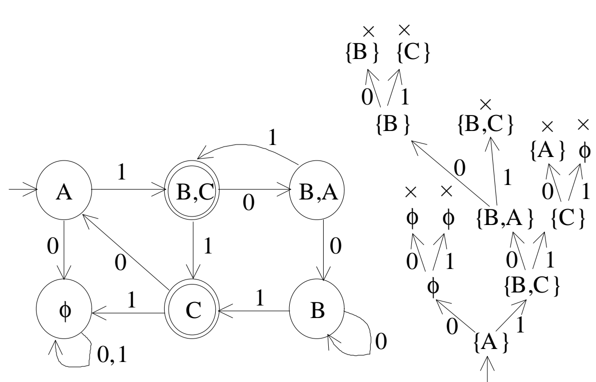
Assignment 1: Set 10/10; In 24/10; Back 26/10

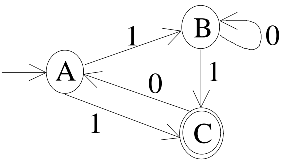
Q: Construct a *deterministic finite-state automaton* which recognises the language $\{a,b\}^*\{aabb\}\{a,b\}^*$. **Test** your solution by tracing through what your machine does to the following strings: (i) aabbabab; (ii) abaaabbbab; and (iii) ababaaabb. A: The transition *diagram* and the *table* are as shown on the right. States: $S = \{1,2,3,4,5\}$; **Initial** State: $s_0 = 1$; **Terminal** States: $F = \{5\}$. (i) ↘aabbabab in state 1; a↘abbabab in state 2; aa↘bbabab in state 3; aab↘babab in state 4; aabb↘abab in state 5; aabba↘bab in state 5; aabbab↘ab in state 5; aabbaba↘b in state 5; and aabbabab↘ in state 5. Because state 5 is a *terminal* state, then the string aabbabab is **accepted** by this machine. (ii) and (iii) are **similar**.



	a	b
→ 1	2	1
2	3	1
3	2	4
4	2	5
○ 5	5	5

Q: Convert the *non-deterministic* automaton shown into a *deterministic* automaton recognising the **same** language, using the "transition tree and subset" *construction*. A: The transition *tree* and "glued" deterministic *diagram* are as shown on the left. You could also if you wanted to provide the <u>states</u>, <u>initial</u> state, <u>terminal</u> states and <u>transition table</u> *information*, in addition to the details of how you used the *subset construction* to go from one **node** to the next in the transition tree.

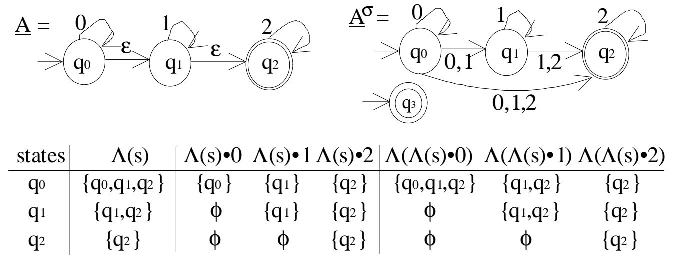




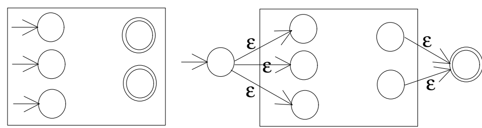
12th October 2000

Let <u>A</u> be an ε-machine. We can construct a *non-det. f.s.a.* <u>without</u> ε-transitions $\underline{A}^\sigma$ s.t. $L(\underline{A}) = L(\underline{A}^\sigma)$. The *details* are as follows: $\underline{A}^\sigma = (S \cup \{q_0\}, \Sigma, s_0 \cup \{q_0\}, \delta_\varepsilon^\sigma, F^\sigma)$, where $F^\sigma = \{F$ if $\varepsilon \notin L(\underline{A})$ (*no* q_0 state), and $F^\sigma = F \cup \{q_0\}$ if $\varepsilon \in L(\underline{A})$ (q_0 *terminal* state)$\}$; for all $a \in \Sigma$, $\delta_\varepsilon^\sigma(q_0,a) = \phi$ (no transitions *out* of q_0); and $\delta_\varepsilon^\sigma(s,a) = \Lambda(\Lambda(s)\bullet a)$, with • in <u>A</u>.

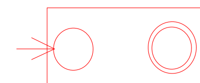
On the *basis* of the argument given last time, we have the following **theorem**: $\underline{A}^\sigma$ is a non-det. *machine* without ε-transitions, and $L(\underline{A}^\sigma) = L(\underline{A})$. *Example*: $\Sigma = \{0,1,2\}$, with $L = \{0\}^*\{1\}^*\{2\}^*$. (If $x \in L$, then $x = uvw$, where $u \in \{0\}^*$, $v \in \{1\}^*$, and $w \in \{2\}^*$, so that $x = 0^m 1^n 2^p$, with $m, n, p \geq 0$). **Example**: see *right*.



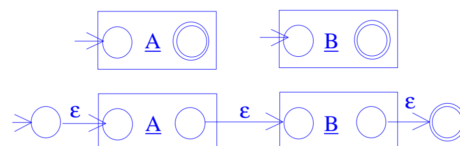
states	Λ(s)	Λ(s)•0	Λ(s)•1	Λ(s)•2	Λ(Λ(s)•0)	Λ(Λ(s)•1)	Λ(Λ(s)•2)
q_0	$\{q_0,q_1,q_2\}$	$\{q_0\}$	$\{q_1\}$	$\{q_2\}$	$\{q_0,q_1,q_2\}$	$\{q_1,q_2\}$	$\{q_2\}$
q_1	$\{q_1,q_2\}$	ϕ	$\{q_1\}$	$\{q_2\}$	ϕ	$\{q_1,q_2\}$	$\{q_2\}$
q_2	$\{q_2\}$	ϕ	ϕ	$\{q_2\}$	ϕ	ϕ	$\{q_2\}$

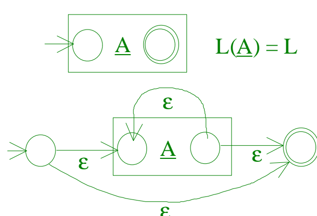


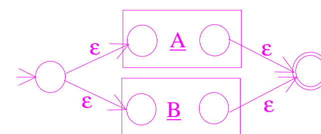
A **normalised** ε-machine is one with a *unique* initial state and a unique *terminal* state, and the only transitions coming out of →O are ε-transitions (and there are *none* coming in), and the only transitions going into ◎ are ε-transitions (and there are *none* coming out). They are pictured as shown in red.



(1) If L and M are *recognisable,* then so too is LM. Let L = L(A), and let M = L(B), where A and B are *normalised*. As you can see from the blue diagram, $\varepsilon l \varepsilon m \varepsilon = lm \in$ LM. We can *convert* this into a deterministic f.s.a., hence LM is recognisable.







(2) L∪M is recognisable. As you can see from the purple diagram, again we can *convert* into a det. f.s.a. which recognises L∪M. (3) If L is recognisable, then L* is recognisable. **Machine** for L*: as shown in green ($L^* = L^0 \cup L \cup L^2 \cup L^3 \cup \ldots$, where $L^0 = \varepsilon$, $L^2 = xy$ (with x, y ∈ L), etc.).

17th October 2000

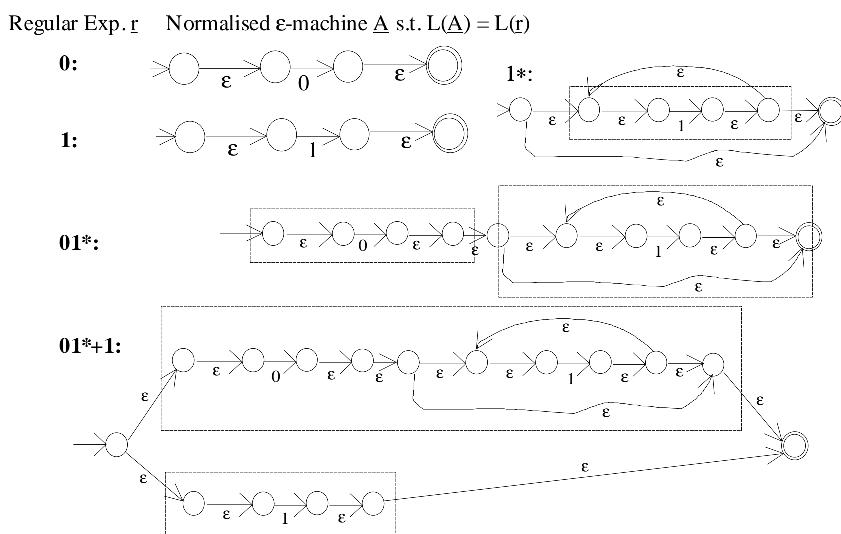
Summary: The following are all *examples* of recognisable languages (Σ = {a,b}): ϕ, {ε}, {a}, {b}. If L and M are *recognisable,* then so too is L∪M; if L and M are recognisable, then so too is LM; and if L is recognisable, then so too is L*. **Example**: The following is recognisable: (({a}*•{b}*)*∪{a}*)*•({a}{b})*.

Idea: Any language which can be *described* by means of an expression using only •, ∪, *, ϕ, {ε}, and {a}, with a ∈ Σ, is *recognisable*. **Regular expressions** make this idea precise. *Definition*: The set of regular expressions over Σ is defined *inductively* as follows: (**R1**) ϕ, ε and a (a ∈ Σ) are all *regular* expressions; (**R2**) If R and S are regular expressions, then so **too** are: (R*), (RS) and (R+S); (**R3**) Every *regular* expression is obtained by applying (**R1**) and (**R2**) a finite number of times. **Example**: ((0(1*))+1) (over Σ = {0,1}). Note that to **omit** brackets, assume that * has *higher precedence* than + and •, and that • has *higher precedence* than +. In general, you need only use brackets to avoid **ambiguity**. Therefore, our example becomes 01*+1. **Idea**: The *interpretation* of 01*+1 is {0}{1}*∪{1}.

Definition: Every regular expression r *describes* a language L(r) obtained in the following way: (**L1**) L(ϕ) = ϕ; L(ε) = {ε}; and L(a) = {a}, where a ∈ Σ; (**L2**) L(R+S) = L(R)∪L(S); L(RS) = L(R)L(S); and L(R*) = L(R)*, where R and S are *regular expressions*. **Example**: L(01*+1) = L(01*)∪L(1) = L(01*)∪{1} = L(0)•L(1*)∪{1} = {0}L(1*)∪{1} = {0}L(1)*∪{1} = {0}{1}*∪{1}.

Definition: A language L is *regular* if there is a regular expression R s.t. L = L(R). **Theorem** *(First Half of Kleene's Theorem)*: Every *regular* language is *recognisable*. **Proof**: Let L = L(r) for some regular expression r. We need to *construct* a machine A s.t. L(A) = L. We prove the theorem by *induction* on the number of operators (i.e. +, •, *) in r. *Base Case:* Number of operators is zero. It follows that r = ϕ, ε or a for some a ∈ Σ. The proof that L(r) is *recognisable* is immediate.

(IH, Inductive Hypothesis): Assume that the result is **true** for all regular expressions r having ≤ n operators. Now prove that the result is true for those *regular expressions* having n+1 operators. Let S have n+1 operators. It follows that there are **3** cases: S = R•T, S = R+T, and S = R*. We *conclude* the proof using the **previous** theorem.



We shall prove that every *recognisable* language is regular. When we have done this, we will have proved **Kleene's Theorem**: A language is recognisable *if and only if* it is regular. An *example* for the first half of Kleene's Theorem is shown **on the left**. Remember that this is not a **practical** algorithm — it merely proves the *idea*.

19th October 2000

Note: From *now on,* we will use + *rather* that ∪, and ε *rather* than {ε}.

Definition: If r and s are *regular* expressions, r = s means that L(r) = L(s). **Some Language Properties**: Let A, B, C, ... ⊆ Σ*. Then (A+B)+C = A+(B+C); A+A = A; (AB)C = A(BC); A(B+C) = AB+AC; (B+C)A = BA+CA; ϕ+A = A = A+ϕ; and εA = A = Aε. **Notation**: $\Sigma_{i=1}^{\infty} A_i = A_1+A_2+A_3+\ldots$ Therefore, $A(\Sigma_{i=0}^{\infty} B_i) = \Sigma_{i=0}^{\infty} AB_i$, and $(\Sigma_{i=0}^{\infty} B_i)A = \Sigma_{i=0}^{\infty} B_iA$.

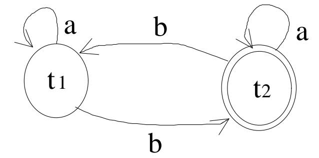
Example: Let Σ = {a,b}, and let X be some Σ-language which satisfies the *following condition*: X = aX+b (X = {a}X∪{b}). Then X ≠ ϕ as {a}ϕ+b = ϕ+b = b, and X ≠ ε as a{ε}+b = a+b. Let us try X = a*b. The **RHS** is a(a*b)+b = a((ε+a+a^2+a^3+...)b)+b = (a+a^2+a^3+...)b+b = ab+a^2b+a^3b+...+b = b+ab+a^2b+a^3b+... = (ε+a+a^2+...)b = a*b.

Let A, E ⊆ Σ*. X = AX+E is called a (right) linear equation for X. **Claim**: X = A*E is a *solution of this equation*. **Proof**. $A(A^*E) + E = A((\Sigma_{i=0}^{\infty} A_i)E) + E = A(\Sigma_{i=0}^{\infty} A_iE) + E = \Sigma_{i=0}^{\infty} A_{i+1}E + E = E + \Sigma_{i=0}^{\infty} A_{i+1}E = E + AE + A^2E + \ldots = (\Sigma_{j=0}^{\infty} A_j)E = A^*E$. **End of proof**. **Theorem (Arden):** Let X = AX+E, where A, E ⊆ Σ*. Then (i) A*E is a *solution*; (ii) If Y is *any* solution, then A*E ⊆ Y, and (iii) If ε ∉ A, then A*E is the **only** solution. **Proof**: see later.

Examples. (i) X = ε+aX+bY (---(1)), Y = bX (---(2)). This is a system of *2 equations in two unknowns,* X and Y. Substitute (2) into (1) ⇒ X = ε+aX+b(bX) = ε+aX+b^2X = ε+(a+b^2)X = (a+b^2)X+ε. By *Arden*, X = (a+b^2)*ε = (a+b^2)*. And so Y = b(a+b^2)*. (ii) X = ε+bX+aY+cZ (---(1)); Y = (b+c)X+aZ (---(2)); and Z = (a+b)X+cY (---(3)). We have Σ = {a,b,c}.

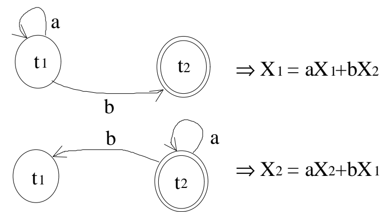
Substitute (3) into (2) and (1) giving Y = (b+c)X + a[(a+b)X+cY] (---(4)) and X = ε+bX+aY + c[(a+b)X+cY] (---(5)). Now Y = (b+c)X + a(a+b)X + acY = [(b+c)+a(a+b)]X + acY; X = ε+bX+aY+c(a+b)X + c^2Y = ε+[b+c(a+b)]X + (a+c^2)Y. From (4), Y = acY + [(b+c)+a(a+b)]X. By **Arden's** Theorem, Y = (ac)*([(b+c)+a(a+c)]X). Now *back* substitute.

Exercise: Solve $X = aX+bY$ (---(1)), $Y = aY+bX+\varepsilon$ (---(2)). Apply *Arden's Theorem* to (2) to get $Y = a^*(bX+\varepsilon)$. Now substitute into (1) to get $X = aX + b(a^*(bX+\varepsilon)) = X(a+ba^*b) + ba^*$. Apply **Arden** to get $X = (a+ba^*b)^*(ba^*)$. Now *substitute* into $Y = a^*bX + a^*$ to get Y.

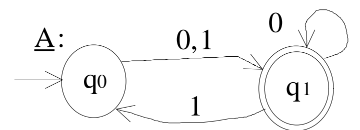


We shall now show how *equations* can be constructed from automata (this will work for **det**. and **non-det**. f.s.a., but **no** ε-automata). For the *example* shown, let $X_1 = L(\underline{A}) = \{x \in \Sigma^*: t_1\bullet x \in F\}$, and let $X_2 = \{x \in \Sigma^*: t_2\bullet X \in F\}$.

Start with X_1. A string in X_1 *begins* with either an "a" or a "b". If $w \in X_1$ *begins* with **a**, then $w = aw'$, where $w' \in X_1$. If $w \in X_1$ *begins* with **b**, then $w = bw''$, where $w'' \in X_2$. So $X_1 = aX_1+bX_2$ (---(1)). Starting with X_2, $X_2 = \varepsilon+aX_2+bX_1$ (---(2)) (ε because t_2 is **terminal**). Now *solve* these equations. Apply **Arden's** theorem to (2) to obtain $X_2 = a^*(\varepsilon+bX_1)$ (---(3)). Substitute (3) into (1) to get $X_1 = aX_1+ba^*(\varepsilon+bX_1)$; $X_1 = X_1(a+ba^*b)+ba^*$. Apply *Arden* to obtain $X_1 = (a+ba^*b)^*(ba^*) = L(\underline{A})$.

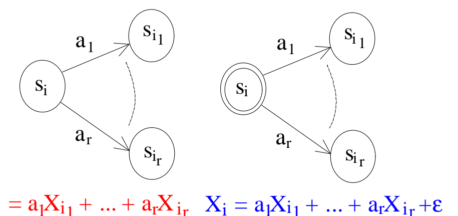


Exercise: From the diagram, we obtain the *equations* $X_0 = 0X_1+1X_1$ and $X_1 = 0X_1+1X_0+\varepsilon$. Apply *Arden* to (2) to get $X_1 = 0^*(1X_0+\varepsilon)$. **Sub.** into (1) to get $X_0 = 00^*(1X_0+\varepsilon)+10^*(1X_0+\varepsilon) = 00^*1X_0+00^*+10^*1X_0+10^* = X_0(00^*1+10^*1) + 00^* + 10^*$. Apply *Arden* to get $X_0 = (00^*1+10^*1)^*(00^*+10^*) = L(\underline{A})$.



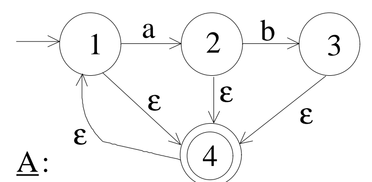
Theorem: Let $\underline{A}$ be any (*non*-) det. f.s.a. without ε-transitions. Then $L(\underline{A})$ is **regular**.
Proof (*det*. case): Let $\underline{A} = (S = \{s_1,\ldots,s_n\}, \Sigma, s_1, \delta, F)$. For each i ($1 \le i \le n$), *define* $X_i = \{z \in \Sigma^*: s_i\bullet z \in F\}$. It is *clear* that $X_1 = L(\underline{A})$.

We now write down the following *right linear equations* ($\Sigma = \{a_1,\ldots,a_r\}$) as shown on the *right*. The **left** hand diagram holds by generalising arguments used in examples. The **right** hand diagram holds since s_i is *terminal*. We have seen that n right linear equations can be **solved** (start with the last equation and work up); no ε's will appear in the languages multiplying the unknowns as we solve them; and so by *Arden's Theorem*, the system has a **unique** solution. In Arden's Theorem, if A and E are *regular*, this implies that A*E is *regular*. Hence X_1 is regular, and we have now proved the *following* theorem (...see after Assignment 2).



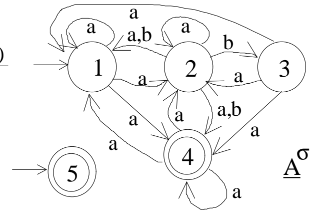
Assignment 2: Set 24/10; In 7/11; Back 14/11

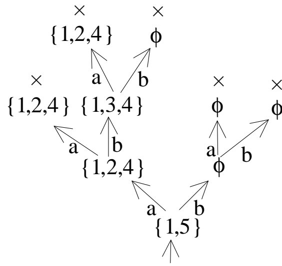
Q: Consider the *non-deterministic* ε-machine shown. (i) Convert $\underline{A}$ into a *non-deterministic* machine **without** ε-transitions $\underline{A}^\sigma$ using the *algorithm* of the lectures; (ii) convert $\underline{A}^\sigma$ into a *deterministic* machine $((\underline{A}^\sigma)^d)^c$ which is *connected*; (iii) write down the **language equations** associated with the machine obtained in (ii); and (iv) **solve** the equations in (iii), and hence find a *regular* expression for $L(\underline{A})$.

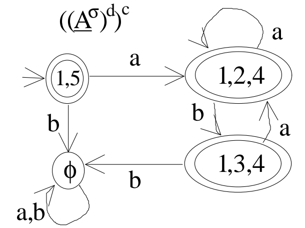


Following the *algorithm*, let us calculate the ε-closures of the **states**, and place them in a table. **Note**: We need *state 5* in $\underline{A}^{\sigma}$ because the *empty* string is accepted by machine $\underline{A}$. (ii) The transition tree is as shown on the left. We use the <u>subset construction</u> to go from *one* node to the *next* in the tree. From the transition tree, we can draw the *(deterministic) transition diagram* as shown. **Note**: State 1,5 is terminal because the **empty** string is accepted by machine $\underline{A}$, and thus has to be accepted by *this* machine too. **States:** S = {15, 124, 134, ϕ}. **Initial State:** s_0 = 15. **Terminal States:** F = {15,124,134}. We can also draw the *transition table.*

State	Λ(s)	Λ(s)•a	Λ(s)•b	Λ(Λ(s)•)	Λ(Λ(s)•b)
1	{1,4}	{2}	ϕ	{2,4,1}	ϕ
2	{2,4,1}	{2}	{3}	{2,4,1}	{3,4,1}
3	{3,4,1}	{2}	ϕ	{2,4,1}	ϕ
4	{4,1}	{2}	ϕ	{2,4,1}	ϕ







(iii) The *language equations* are $X_{15} = aX_{124}+bX_{\phi}+\varepsilon$ (---(1)); $X_{124} = aX_{124}+bX_{134}+\varepsilon$ (---(2)); $X_{134} = aX_{124}+bX_{\phi}+\varepsilon$ (---(3)); and $X_{\phi} = aX_{\phi}+bX_{\phi}$ (---(4)). (**Simplify** the equations if possible). (iv) (4) ⇒ $X_{\phi} = (a+b)X_{\phi}$. Apply *Arden* to (4) and we get $X_{\phi} = (a+b)^*(\phi) = \phi$. So *we obtain* $X_{15} = aX_{124}+\varepsilon$ (---(5)); $X_{124} = aX_{124}+bX_{134}+\varepsilon$ (---(6)); and $X_{134} = aX_{124}+\varepsilon$ (---(7)).

Apply *Arden* to (6) so that $X_{124} = a^*(bX_{134}+\varepsilon)$ (---(8)). Sub. for X_{124} in (7) to get $X_{134} = a(a^*(bX_{134}+\varepsilon))+\varepsilon = aa^*bX_{134}+aa^*+\varepsilon$ (---(9)). Apply *Arden* to (9) so that $X_{134} = (aa^*b)^*(aa^*\varepsilon)$ (---(10)). Sub. for X_{134} from (10) to (8) to obtain $X_{124} = a^*(b(aa^*b)^*(aa^*+\varepsilon)+\varepsilon)$ (---(11)). <u>Finally</u>, sub. for X_{124} *from* (11) *into* (5) so that $X_{15} = a(a^*(b(aa^*b)^*(aa^*+\varepsilon)+\varepsilon))+\varepsilon = aa^*b(aa^*b)^*(aa^*+\varepsilon)+aa^*+\varepsilon = L(\underline{A})$.

<u>Kleene's Theorem</u>

A language is *recognisable* if and only if it is **regular**. Let us now go back to <u>**Arden's Theorem**</u>: Let X = AX+E, where A, E ⊆ Σ*, then (i) A*E is a *solution*; (ii) If Y is **any** solution, then A*E ⊆ Y; and (iii) If ε ∉ A, then A*E is the **unique** solution. <u>**Proof of (ii)**</u>: Let Y be *any* solution to X = AX+E, i.e. <u>Y = AY+E</u>. Here, AY ⊆ Y, and E ⊆ Y. Now E ⊆ Y ⇒ AE ⊆ AY (*think*). But AY ⊆ Y, and so AE ⊆ Y. Also, $A^2E = A(AE) \subseteq AY \subseteq Y$. So E ⊆Y, AE ⊆ Y, $A^2E \subseteq Y$, etc. By *induction*, $A^nE \subseteq Y$ for *all* $n \geq 0$. But $A^*E = (\Sigma_{n=0}^{\infty} A^n)E = \Sigma_{n=0}^{\infty} A^nE \subseteq Y$, i.e. A*E ⊆ Y.

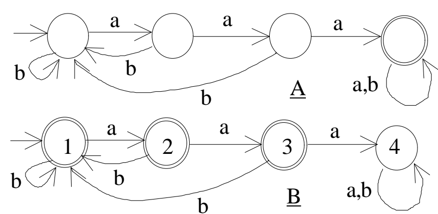
<u>**Proof of (iii)**</u>: Let W be *any solution* to our equation. By (ii), we <u>know that</u> A*E ⊆ W. To prove that in fact we have A*E = W, we have to *show* that W\A*E = ϕ. We shall prove this by **contradiction** — <u>by supposing that W\A*E ≠ ϕ</u>. Let z ∈ W\A*E be of *minimal* length. By **definition**, z ∈ W and z ∉ A*E. **Now** A*E⊇ E. Thus z∉ E. By *assumption*, W = AW+E. So z ∈ W ⇒ z ∈ AW+E. It *follows* that Z ∈ AW since z ∉ E. Therefore, z = xy, *where* x ∈ A and y ∈ W. By <u>assumption</u>, ε ∉ A, so that $|x| \geq 1$. <u>It **follows** that</u> |y| < |z|, with y ∈ W. <u>CLAIM</u>: y ∉ A*E. Suppose to the *contrary* that y ∈ A*E. Then z = xy ∈ A(A*E) ⊆ A*E. But this **can't** happen, because <u>z ∈ W\A*E</u>. We have *shown* that y ∈ W\A*E, and that |y| < |z|, which is a <u>contradiction</u>. It *follows* that W\A*E = ϕ, and **hence** W = A*E as *required.*

Exercises

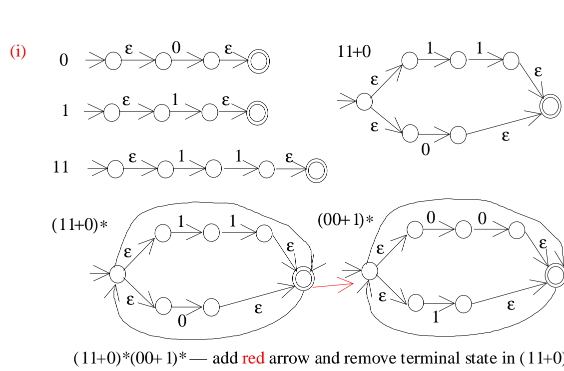
Q: Construct a *regular expression* for each of the following languages over $\Sigma = \{a,b\}$: (i) All strings in which *a* always appears in **multiples of 3**; (ii) all strings which contain *exactly 3 b's*; (iii) all strings which contain *exactly 2 or 3 b's*; (iv) all strings which **end** in a *double letter*; (v) all strings which have **exactly one** double letter (hard); (vi) all strings which do **not** contain *aaa* as a factor; and (vii) all strings in which the **total** number of *a's* is divisible by 3.

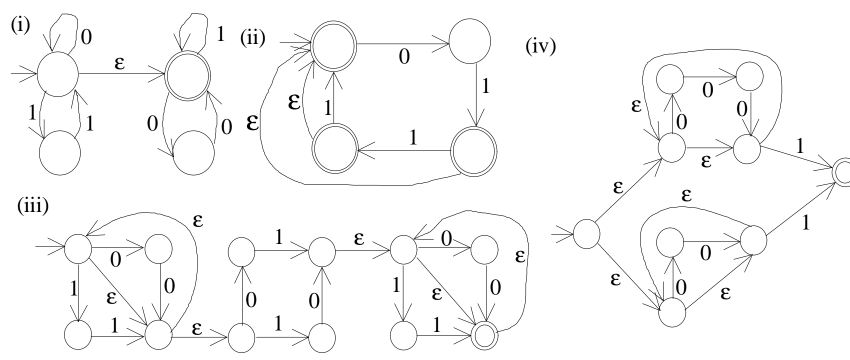
A: (i) $(a^3+b)^*$. (ii) $a^*ba^*ba^*ba^*$. (iii) $a^*ba^*ba^* + a^*ba^*ba^*ba^*$. (iv) $(a+b)^*aa+ (a+b)^*bb$. (v) Let's consider first the case where the *double letter* is aa. The strings we want are of the **form** (no double letters not ending in a), aa, (no double letters doesn't begin with a). A string with no double letters must have letters which *alternate*. These are of the form $(\varepsilon+b)(ab)^*(\varepsilon+a)$. Thus the strings where the double letter is an aa *must* be of the form $(\varepsilon+b)(ab)^*aa(b(ab)^*(\varepsilon+a)+\varepsilon)$. This **can** be simplified. Observe that $b(ab)^* = (ba)^*b$. Thus we get $(\varepsilon+b)(ab)^*aa((ba)^*b(\varepsilon+a)+\varepsilon)$ which is *just* $(\varepsilon+b)(ab)^*aa(ba)^*(\varepsilon+b)$. Hence the *regular expression* we require is $(\varepsilon+b)(ab)^*aa(ba)^*(\varepsilon+b) + (\varepsilon+a)(ba)^*bb(ab)^*(\varepsilon+a)$.

(vi) $(b+ab+a^2b)^*(\varepsilon+a+a^2)$. Tip: If a machine *recognises* r, then to get a machine that recognises $\Sigma^*\backslash r = s$, swap the **terminal** and the **non-terminal** states. So here you can write down the machine which recognises all strings *containing* aaa as a factor, and modify it. The language of all {a,b} strings which do contain aaa as a factor is accepted by the *first machine*, A. Thus a machine which recognises those strings which do not contain aaa as a factor is B. The language expressions we get are $X_1 = bX_1+aX_2+\varepsilon$, $X_2 = \ldots$ which we solve by **Arden** and **substitution** to obtain the *regular* expression $(b+ab+a^2b)^*(a^2+a+\varepsilon)$. (vii) $(b^*ab^*ab^*ab^*)^*$.



Q: Construct ε-machines which *recognise the languages* associated with the following regular expressions. Try to make them as **simple** as possible, and note that you do not have to follow the "*normalised **e**-machine*" approach of the lectures. (i) $(11+0)^*(00+1)^*$; (ii) $(01+011+0111)^*$; (iii) $(00+11)^*(01+10)(00+11)^*$; (iv) $(000)^*1+(00)^*1$. A: As shown on the left. Try to **remove** as much ε-edges as is possible.





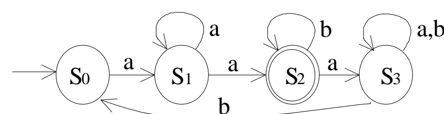
Q: Prove that the **following** equalities hold: (i) $(a^*b)^*a^* = (a+b)^*$; (ii) $a(ba)^* = (ab)^*a$; and (iii) $a^* = (aa)^*+a(aa)^*$. A: (i) Because $(a+b)^*$ consists of *all strings over* {a,b}, it is obvious that $(a^*b)a^* \subseteq (a+b)^*$. We need to prove the **reverse** inclusion.

Let $x \in (a+b)^*$. If x consists entirely of a's, then it is *clearly* in the LHS. Otherwise, x consists of at least one b and some a's. Here, we can **write** $X = wa^n$, where $n \geq 0$ and w *ends* in b. Thus $w = a^{m_1}ba^{m_2}b....a^{m_p}b$, where $m_i \geq 0$ and $1 \leq i \leq p$. (*<u>Idea</u>*: Let the b's serve as <u>dividers</u> in the string). But then $w = (a^{m_1}b)(a^{m_2}b)...(a^{m_p}b) \in (a^*b)^*$. *Thus* $x \in (a^*b)^*a^*$.

(ii) A **typical** element of $a(ba)^*$ is $a(ba)^n$, where $n \geq 0$. If $n = 0$, then $a(ba)^0 = a \in (ab)^*a$. *Otherwise*, $a(ba)^n = a(ba)(ba)...(ba)$ ((ba) *n times*). Therefore, $a(ba)^n = (ab)(ab)...(ab)a = (ab)^na$. Hence $a(ba)^n \in (ab)^*a$, *and* $a(ba)^* \subseteq (ab)^*a$. A **symmetric** argument in the other direction shows that $(ab)^*a \subseteq a(ba)^*$. Hence the *proof* is complete. (iii) An element of a^* has either **odd** or **even** length. The result *follows*.

Q: *Check* that $X_1 = (a+bb)^*$ and $X_2 = b(a+bb)^*$ are **solutions** to $X_1 = \varepsilon+aX_1+bX_2$ (---(1)) and $X_2 = bX_1$ (---(2)). (*Substitute into the RHS* and check that you *get* the LHS). A: Substitute the solution into the RHS of (1): $X_1 = \varepsilon+a(a+bb)^*+b(b(a+bb)^*) = \varepsilon+(a+bb)(a+bb)^*$. Now $(a+bb)(a+bb)^* = \Sigma_{n=0}^{\infty} (a+bb)(a+bb)^n = \Sigma_{n=0}^{\infty}(a+bb)^{n+1} = \Sigma_{n=1}^{\infty}(a+bb)^n$. But $\varepsilon = (a+bb)^0$, so $\varepsilon+\Sigma_{n=1}^{\infty}(a+bb)^n = \Sigma_{n=0}^{\infty}(a+bb)^n$. So $X_1 = (a+bb)^*$. <u>Fine so far</u>. Now substitute *the solution* into the RHS of (2): $X_2 = b((a+bb)^*) = b(a+bb)^*$. <u>QED</u>.

Q: Write down the *equations* associated with the machine shown, and hence find the **language** it accepts. A: We have $X_0 = aX_1$ (---(1)), $X_1 = aX_1+aX_2$ (---(2)), $X_2 = \varepsilon+aX_3+bX_2$ (---(3)), and $X_3 = aX_3+bX_3+bX_0$ (---(4)).



(4) $\Rightarrow X_3 = (a+b)X_3+bX_0$. Apply *Arden* to get $X_3 = (a+b)^*(bX_0)$ (---(5)). Sub. (5) into (3) to get $X_2 = \varepsilon+a(a+b)^*(bX_0) + bX_2$. Apply *Arden* to get $X_2 = b^*[\varepsilon+a(a+b)^*bX_0]$ (---(6)). Sub. (6) into (2) for X_2 to <u>get</u> $X_1 = aX_1+ab^*[\varepsilon+a(a+b)^*bX_0]$. Apply *Arden* to get $X_1 = aa^*b^*[\varepsilon+a(a+b)^*bX_0]$ (---(7)). Sub. (7) into (1) for X_1 to <u>get</u> $X_0 = aaa^*b^*[\varepsilon+a(a+b)^*bX_0] = a^2a^*b^*a(a+b)^*bX_0 + a^2a^*b^*$. Apply *Arden* to get $X_0 = (a^2a^*b^*a(a+b)^*b)^*a^2a^*b^* = L(\underline{A})$.

31st October 2000

<u>2. Context Free Languages and Pushdown Automata</u>

<u>General, Phrase-structure Grammars</u>

Recognisable languages and their automata (f.s.a.) are too limited to cope with what we might genuinely regard as languages: (1) High-level programming languages <u>(Pascal, C, Java, etc.)</u>; (2) Natural Languages <u>(English, Welsh, Russian, etc.)</u>; and (3) Mathematical Languages <u>(propositional logic, predicate logic, etc.)</u>.

<u>**Background**</u>. Basic question: How can a *natural* language such as English with an **infinite** number of possible sentences be generated by **finite** means? Evidence suggests that languages are *rule driven*. Language is generated by a *finite* number of rules applied to a finite vocabulary.

Example: **The cat sat on the mat**. **'Cat'** and **'Mat'** are Nouns. **'Sat'** is a Verb. **'On'** is a preposition. **'The'** is a determinate. Each word in the language belongs to a *grammatical category*. The structure of sentences is determined by the way grammatical categories can be combined. Language has the following *ingredients*: Grammatical categories (which you don't see) and words i.e. symbols which make up the *sentences*. (Two levels).

Example 1: *A fragment of English grammar*. There are the following grammatical categories (we'll usually use the **latter** phase): Sentence ("start symbol") **S**; Noun-phrase **NP**; Verb-phrase **VP**; Noun **N**; Definite Article **T**; and Verb **V**. There are the following productions (these are rules for *rewriting certain strings* as certain other strings):

(1) **S** → **NP VP**; (2) **NP** → **T N**; (3) **VP** → **V NP**; (4) **T** → the. (5) **N** → man | ball | ... ; and (6) **V** → hit | took | ... The symbols *"the", "man", "ball", "hit"*, etc. are called terminals. It is important to note that we are thinking of the words "the", "man", etc. as being *individual symbols*, not strings. The symbol "|" is read as "**or**", and is a way of combining several productions having the same left-hand side.

We shall give a more *formal* definition later, but for now a (phrase structure) grammar is a 4-tuple **G** = (N, Σ, P, S) consisting of a finite set of *non-terminals* N, a finite set of *terminals* Σ, a finite set of *productions* P, and a *start symbol* S. Example of *generating a sentence* using the above grammar: **S** →(1) **NP VP** →(2) **T N VP** →(3) **T N V NP** →(2) **T N V T N** →(4) the **N V T N** →(5) The man **V T N** →(6) The main hit **T N** →(4) The man hit the **N** →(5) The man hit the ball. In the case of *example* 1, N = {**S**, **NP**, **VP**, **N, T**, **V**}, Σ = some subset of all possible *English* words, P = (1)-(6), and **S** = **S**.

Example 2. Consider the grammar **G** = ({S}, {a,b}, P, S), where P has the *following productions:* (1) S → aSb, (2) S → ab. Examples of strings in Σ* generated by *this grammar*: S →(2) ab; S →(1) aSb →(2) aabb = a^2b^2; and S →(1) aSb →(1) aaSbb →(1) aaaSbbb →(2) aaaabbbb = a^4b^4. In the last case, S $\rightarrow^*$ a^4b^4 ("$\rightarrow^*$" means a *sequence* of →'s).

You can prove that (1) S $\rightarrow^*$ a^nb^n, where $n \geq 1$, and (2) **only** such strings can be generated. L(G) = {w: w ∈ Σ*, S $\rightarrow^*$ w} is called the *language* of the grammar. Here, L(G) = {a^nb^n: $n \geq 1$}. We know that this language is *not* recognisable, and thus these grammars are more **powerful** than f.s.a.

Example 3: the following is a grammar for a fragment of a *Pascal-type language* called the "language of **while**-programs": the grammar **G** = (V, X, P, C) is given by V = {C, S, S_1, S_2, A, W, C, U, T}, and X consists of **begin**, **end**, **pred**, **succ**, **while** and **do**, together with the following symbols: :=, ≠, ;, (,), 0, x, and y.

The *productions* are as follows: (1) C → **begin**S_1**end** (2) S_1 → SS_2 (3) S_2 → ;S_1|ε. (4) S → A|W|C (5) A → V:=T (6) T → **pred**(V)|**succ**(V)|0 (7) W → **while**V≠V **do** S (8) V → x|y. *Derivation*: we can use it to generate **begin** y:=0; **while** x≠y do y:=**succ**(y) end. Construction: C →(1) **begin**S_1**end** →(2) **begin** SS_2 **end** →(4) **begin** A S_2 **end** →(5) **begin** V:=T S_2 **end** →(8) **begin** y:=T S_2 **end** →(6) **begin** y:=0 S_2 **end**, etc.

Example 4: Consider the grammar **G** = ({A,S}, {a,b,c}, P, S), where P has the *following productions:* (1) S→abASc|ε; (2) bAa→abA; (3) bAc→bc; and (4) bAb→bbA. **Exercise**: Derive $a^3b^3c^3$. *Now* S ⇒(1) abASc ⇒(1) abAabAScc ⇒(1) abAabAabASccc ⇒(1) $(abA)^3c^3$ = $(abA)^2abAcc^2$ ⇒(3) $(abA)^2abcc^2$ = $(abA)abAabc^3$ ⇒(2) $(abA)aabAbc^3$ ⇒(4) $(abA)a^2b^2Ac^3$ = $(abA)a^2bbAcc^2$ ⇒(3) $(abA)a^2bbcc^2$ = $(abA)a^2b^2c^3$ = $abAaab^2c^3$ ⇒(2) $aabAab^2c^3$ ⇒(2) $a^2abAb^2c^3$ = a^3bAbbc^3 ⇒(4) $a^3b^2Abc^3$ = a^3bbAbc^3 ⇒(4) $a^3b^3Ac^3$ = $a^3b^2bAcc^2$ ⇒(3) $a^3b^3c^3$.

2nd November 2000

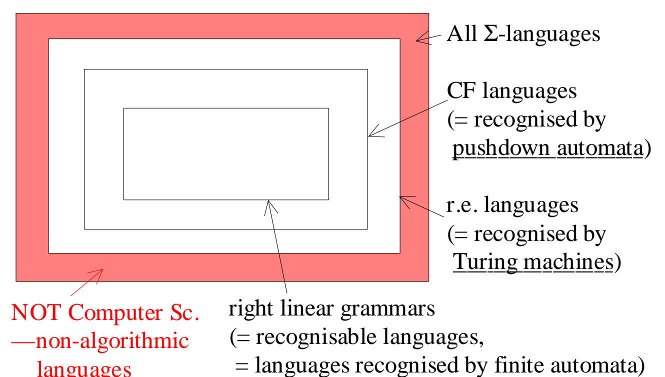
Examples 1 to 3 have rules of the form *single non-terminal* ® *string*. Let us now **formalise** the notion of a grammar. **Definition**: A phrase structure grammar is a *4-tuple* G = (N, Σ, P, S) consisting of (1) a finite set N of *non-terminals*; (2) a finite set Σ of *terminals*; (3) we have N∩Σ = ϕ; (4) P is a finite set of *productions*, where each production is an ordered pair (α,β) (we usually *write* this as α→β, where α, β ∈ (N+Σ)*) **and** α contains at least *one* non-terminal symbol; and (5) S ∈ N is the *start* symbol.

We want to define now the language *generated* by the grammar. **Suppose** that α→β ∈ P, and let w = φαψ and let w' = φβψ. (These belong to (N+Σ)*). We **say** that w' is *immediately* derived from w in G, and write w ⇒ w'. (Note: w = φαψ ⇒ w' = φβψ. (α changes to β)). If $w_1,...,w_n$ is a sequence of strings in (N+Σ)* such that $w_1 \Rightarrow w_2 \Rightarrow w_3 \Rightarrow ... \Rightarrow w_n$, then we say that w_n is *derivable* from w_1, and write $w_1 \Rightarrow^* w_n$.

The *sequence* $w_1,...,w_n$ is called a derivation of w_n from w_1 w.r.t. the grammar G. The language *generated* by G is L(G) = {w ∈ Σ*: S $\Rightarrow^*$ W}. We now **single out** two special classes of grammar. *Definition*: A grammar G = (N, Σ, P, S) is said to be *context-free* (CF) if every production has the **following** form: A→α, where A ∈ N and α ∈ (N+Σ)*. (A *single* non-terminal symbol on the left side of →).

A language L is said to be **context-free** if there is a CF-grammar G such that L = L(G). *Example 4* is **not** a context free grammar, but examples 1 to 3 **are** CF-grammars. (The language in example 4 is $\{a^nb^nc^n: n \geq 0\}$). A language L is *said* to be recursively enumerable (r.e.) if there is a phrase-structure grammar G s.t. L = L(G).

A grammar G is said to be *right linear* if each production in P has the form A→bC, A→b, or A→ε (A,C ∈ N, b ∈ Σ). Let Σ be a fixed alphabet. Then we have the *diagram* as shown, a hierarchy of languages known as the **Chomsky Hierarchy**.



Phrase Structure Grammars

(1) r.e. languages: *productions* $\alpha \to \beta$, where α contains *at least one* non-terminal. (2) CF languages: *productions* $\alpha \to \beta$, where α is a non-terminal. (3) Right-linear languages: *productions* $\alpha \to \beta$, having the *following* forms: $A \to bC$, $A \to b$, and $A \to \varepsilon$, where $A, C \in N$, and $b \in \Sigma$. *Example* (of a right linear grammar and language): G: $S \to 1B|1$ (**start** symbol S), $A \to 1B|1$, $B \to 0A$. Example of a *derivation* in this grammar: $S \Rightarrow 1B \Rightarrow 10A \Rightarrow 101B \Rightarrow 10101A \Rightarrow 10101$. It is possible to show that L(G) = 1(01)*. We shall *show* that this is typical.

Lemma 1: Let G = (N, Σ, P, S) be a *right linear grammar*. Then we can always assume that the rules which **appear** in P have the following forms: $A \to bC$ or $A \to \varepsilon$, i.e. NOT $A \to b$. **Proof**. Suppose that we have *productions* in P of the form $A \to b$. Introduce a new *non-terminal symbol* D ($D \notin N$) and *replace* $A \to b$ by $A \to bD$ and $D \to \varepsilon$. Do this for all unwanted productions, and the *new* grammar G' satisfies the **conditions** of the lemma, **and** L(G) = L(G').

Returning to our example of a *right linear grammar,* G: $S \to 1B$, $A \to 1B$, $B \to 0A$, $S \to 1$, and $A \to 1$, we can *replace* $S \to 1$ by $S \to 1D$ and $D \to \varepsilon$, and **replace** $A \to 1$ by $A \to 1D$. We obtain a *new* grammar G': $S \to 1B$, $A \to 1B$, $B \to 0A$, $S \to 1D$, $A \to 1D$, and $D \to \varepsilon$.

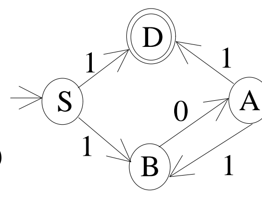
Theorem 2: A language is recognised *by a f.s.a.* if and only if it is generated by a right linear grammar. **Proof**. Let G = (N, Σ, P, S) be a *right-linear grammar* (in the form given by Lemma 1). We shall construct a non-det. automaton A such that L(G) = L(A), where A = (Q, Σ, s_0, δ, F) (**an** initial state s_0). Let Q = N *(states labelled by non-terminals)*, let $s_0 = S$, let $F = \{A: A \in N, A \to \varepsilon \in P\}$, and let δ: (A)—b→(C) be a *transition* in A $\Leftrightarrow A \to bC \in P$.

Claim: L(A) = L(G). **Proof**. Let $w = a_1 \ldots a_n \in$ L(G). This happens $\Leftrightarrow S \Rightarrow^* w \Leftrightarrow S \Rightarrow a_1A_1 \Rightarrow a_1a_2A_2 \Rightarrow \ldots \Rightarrow a_1\ldots a_nA_n \Rightarrow a_1\ldots a_n \Leftrightarrow \to(S)\text{—}a_1\to(A_1)\text{—}a_2\to(A_2)\text{——}\to\ldots\text{—}a_n\to((A_n)) \Leftrightarrow w \in$ L(A). **Converse**: Let A = (Q, Σ, s_0, δ, F) be a **det**. f.s.a. *Define* G = (N, Σ, P, S) as follows: N = Q, S = s_0, P = $\{A \to \varepsilon \Leftrightarrow A \in F;\ A \to cB \Leftrightarrow A \bullet c = B$ in A$\}$. **Claim**: L(G) = L(A). **Proof**: $w = a_1\ldots a_n \in$ L(A) $\Leftrightarrow s_0 \bullet w \in F \Leftrightarrow \to(s_0)\text{—}a_1\to(s_1)\text{—}a_2\to(s_2)\text{——}\to \ldots. \text{—}a_n\to((s_n)) \Leftrightarrow S \to a_1S_2$, $S_1 \to a_2S_2, \ldots, S_{n-1} \to a_nS_n, S_n \to \varepsilon \Leftrightarrow S \Rightarrow^* w \in$ L(G).

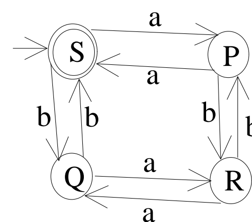
Example: (from *grammar* to *machine* and then from *machine* to *grammar*).

(From grammar to machine)

$S \to 1B$	(S,1,B)
$S \to 1D$	(S,1,D)
$A \to 1B$	(A,1,B)
$B \to 0A$	(B,0,A)
$A \to 1D$	(A,1,D)
$D \to \varepsilon$	((D))



(From machine to grammar)



G = ({S,P,Q,R}, {a,b}, *P*, S)

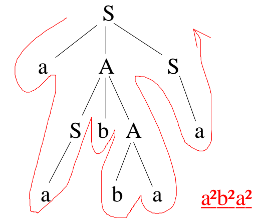
$S \to aP$	$S \to bQ$
$P \to aS$	$P \to bR$
$Q \to aR$	$Q \to bS$
$R \to aQ$	$R \to bP$
$S \to \varepsilon$	

Context Free Languages

Backus-Naur Form (BNF) is nothing more than an *alternative* notation for writing down CF grammars, where non-terminals are written as *descriptions* enclosed in <, > e.g. <description>; → is written as ::=; and | means "*or*" as before. Example: *Grammar 3* becomes easier to read if we use BNF: C ➤ <program>; S_1 ➤ <statement sequence>. So (1) *becomes* <program> ::= begin <statement sequence> end.

9th November 2000

Context Free Languages: Derivation Trees (Parse Trees)

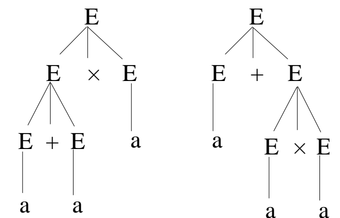


For *CF Grammars,* there is a way of **picturing** derivations. **Example**: Consider G = ({S,A}, {a,b}, P, S), where P: S→aAS|a, A→SbA | SS | ba. For the derivation S ⇒ aAS ⇒ aAa ⇒ aSbAa ⇒ aSbbaa ⇒ aabbaa = $a^2b^2a^2$, the derivation tree is as *shown,* where the $a^2b^2a^2$ read off the tree is the **yield** of the tree.

The *derivation tree* does not capture the order in which we apply the productions — but this doesn't matter. The derivation tree provides the **basis** for assigning a *meaning* to the string derived (i.e. the yield of the tree). *Given* a derivation tree, find examples of derivations with this tree. We can use the derivation tree to **construct** a derivation of $a^2b^2a^2$ which rewrites the left most non-terminal at each stage. S ⇒ aAS ⇒ aSbAS ⇒ aabAS ⇒ aabbaS ⇒ aabbaa.

Right most derivation — rewrite the **right** most terminal at each stage. S ⇒ aAS ⇒ aAa ⇒ aSbAa ⇒ aSbbaa ⇒ aabbaa. This example provides the *idea* behind the proof of the following result: **Proposition**: Let G be a context free grammar. Then w ∈ L(G) ⇔ w has a *left most derivation* ⇔ w has a *right most derivation*. **Proof (Sketch)**. If w has a *left* most derivation, then clearly w ∈ L(G).

Conversely, suppose that w ∈ L(G). Then w has a *derivation*, and so we can construct a derivation tree. But then it is **clear** that we can construct the tree by means of a *left most derivation,* and hence we can derive w in a **left** most way. There is a similar argument for **right** most derivations. Derivation trees / Parse trees. If G is a grammar, if we have terminals Σ, and if w ∈ Σ*, then the process of trying to find a *derivation* tree is called parsing (or *syntax analysis*).



Ambiguity. A CF-grammar is said to be *ambiguous* if there is a string in the language with **more** than one derivation tree. **Example**: Consider G: E→E+E | E×E | (E) | a. For *a*+*a*×*a*, we have *two* possible trees as shown. This is an *ambiguous* grammar. The two different derivation trees could be used to assign two different **meanings** to the string a+a×a.

Compilers. (1) *Lexical Analysis* of a program in a high-level language identifies the *tokens* of the language using finite automata. In Example 3, the tokens are begin, end, pred, succ, while, do; :=, ≠, ;, (,), 0, x, y, ... (2) *Syntax Analysis* parses (i.e. finds a description of) the program.

The Pumping Lemma for CF Languages

The *pumping lemma* describes a property possessed by **all** CF languages. We shall then show that there are languages which do not have the pumping property — and so are not CF. First, some *terminology*: (1) We are dealing with **directed rooted trees** — the implied direction of each edge is *down*, and the root is the vertex at *the* top from which all paths lead from. The vertices (terminals) at the end are the leaves of the tree.

(2) Given a *derivation* tree for a string in a CF grammar, we define the **length** of a path from the root to a terminal symbol to be the number of *non-terminals* on that path. The **height** of a derivation tree is the size of a *maximum length* path.

10th November 2000

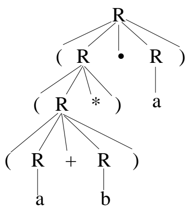
Exercises

Q: Let G = ({S,B,C}, {a,b,c}, P, S), where P consists of the *following* productions: (1) S→aSBC, (2) S→abC, (3) bB→bb, (4) bC→bc, (5) CB→BC, and (6) cC→cc. Show that $a^2b^2c^2$ ∈ L(G), and *describe* L(G). A: S ⇒ aSBc by 1 ⇒ aabCBC by 2 ⇒ aabBCC by 5 ⇒ aabbCC by 3 ⇒ aabbcC by 4 ⇒ aabbcc by 6. L(G) = $\{a^nb^nc^n: n \geq 1\}$. (Get this by *doing examples).*

Q: Let G = ({S,C}, {a,b}, P, S), where P consists of the *following* productions: S→aCa, C→aCa, and C→b. Show that a^3ba^3 ∈ L(G), and *describe* L(G). A: S ⇒ aCa ⇒ aaCaa ⇒ aaaCaaa ⇒ aaabaaa. L(G) = $\{a^nba^n: n \geq 1\}$.

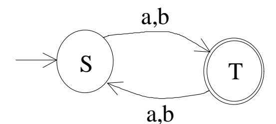
Q: Let G = ({S,A,B,C}, {a,b}, P, S), where P consists of the *following* productions: S→aS, S→aB, B→bC, C→aC, and C→a. **Show** that a^3ba^2 ∈ L(G), and *describe* L(G). A: S ⇒ aS ⇒ aaS ⇒ aaaB ⇒ aaabC ⇒ aaabaC ⇒ aaabaa. L(G) = $\{a^nba^m: n,m \geq 1\}$.

Q: Find *CF grammars* for each of the **following** languages, where Σ = {a,b}: (i) $\{a^nb^{2n}: n \geq 0\}$; (ii) $\{a^nb^{n+2}: n \geq 0\}$; (iii) the *palindromes* of even length; (iv) the *palindromes* of odd length; and (v) *any* palindrome. (A palindrome is a string which reads the same **forwards** as **backwards**. A: (i) S→aSb²|ε. (ii) S→aTb|b², T→b²|aTb. (iii) S→aSa|bSb|ε. (iv) S→aSa|bSb|a|b. (v) S→aSa|bSb|a|b|ε.

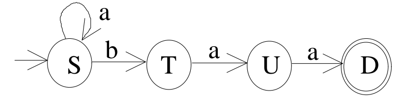


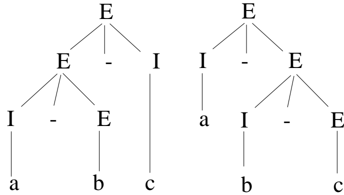
Q: Write a CF grammar for *regular* expressions over Σ = {a,b} (include *all* brackets). Find a derivation tree for (((a+b)*)a). A: R→ a | b | ε | ϕ | (R.R) | (R+R) | (R*). The *derivation tree* is as shown on the left.

Q: Find a *right linear grammar* which generates the language recognised by the **shown** machine. A: G = ({S,T}, {a,b}, P, S), *where* P = {S→aT, S→bT, T→aS, T→bS, T→ε}.



Q: Find a non-det. machine which recognises the language *generated* by the following right linear grammar: G = ({S,T,U}, {a,b}, P, S), where P consists of the *following* productions: S→aS, S→bT, T→aU, and U→a. A: We first have to *convert* the grammar into a suitable form: **replace** U→a by U→aD and D→ε *Therefore,* G = ({S,T,U,D}, {a,b}, P, S), where P = {S→aS, S→bT, T→aU, U→aD, D→ε}. We thus *obtain* the diagram on the right.





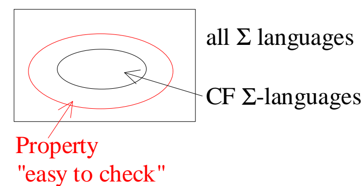
Q: Show that the *following* grammar is **ambiguous** by finding two different derivation trees of a-b-c, where G = ({E,I}, {a,b,c,d,-}, P, E), and where P has the *following* productions: E→I, E→I-E, E→E-I, and I→a|b|c|d. A: See the diagram *shown* on the left.

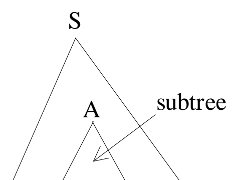
Q: How could the grammar in *Example 1* be written using BNF? A: <sentence> ::= <noun phrase><verb phrase>, <noun phrase> ::= <def. article><noun>, <verb phrase> ::= <verb><noun phrase>, <def. article> ::= the, <noun> ::= man | ball | ...., and <verb> ::= hit | took | ....

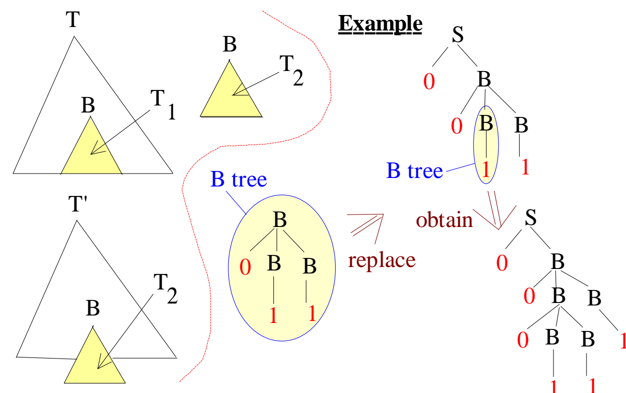
14th November 2000

Proving that there are Languages which are not CF

Subtree replacement principle (only applies to CF grammars). A *subtree* of a derivation tree is a particular vertex of the tree, together with that part of the derivation tree which lies below it. It looks like a **derivation** tree, except that its root can be any non-terminal. If the root is A (∈ N), then the subtree is called an A-tree.



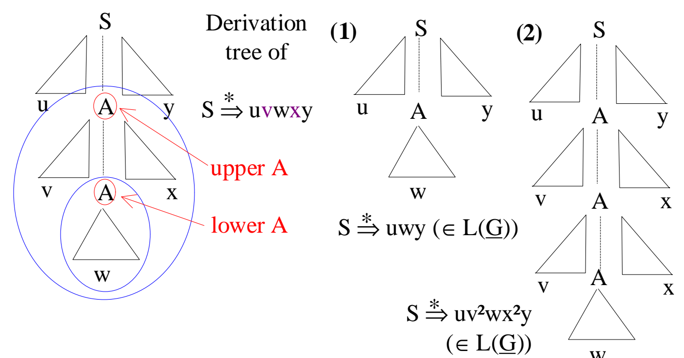




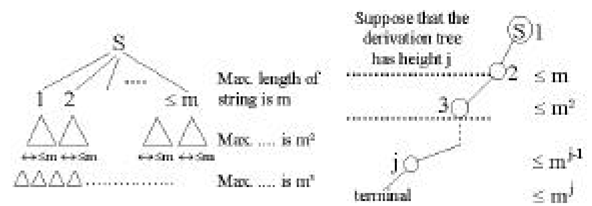
(**SRP**) Let T be a derivation tree, and let T_1 and T_2 be any B-trees. Suppose that T_1 is a subtree of T, then the *tree* T' **obtained** from T by replacing T_1 by T_2 is **also** a derivation tree.

The pumping property for CF languages

We apply the *SRP* to the first derivation tree — concentrate on the **A-trees** (upper & lower). (1) *Replace* the upper A-tree by the lower one. (2) Replace the *lower* A-tree by the upper tree. This process (*replacing* lower by upper) can be **iterated**. If uvwxy ∈ L(G) as above, then uv^iwx^iy ∈ L(G) (for *all* i ≥ 0). We now deal with the question of **derivations** with repeated non-terminals.



Lemma 1: Let G be a CF grammar. Suppose that the *longest* right hand side of any production has **m** symbols. Let T be a *derivation tree* for a string w ∈ L(G). If the **height** of T is j, then $|w| \le m^j$. **Proof (Sketch): shown in the diagram.**

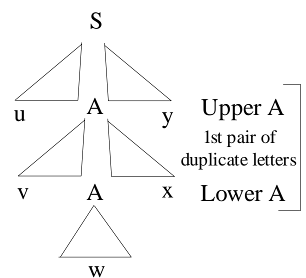


The Pumping Lemma

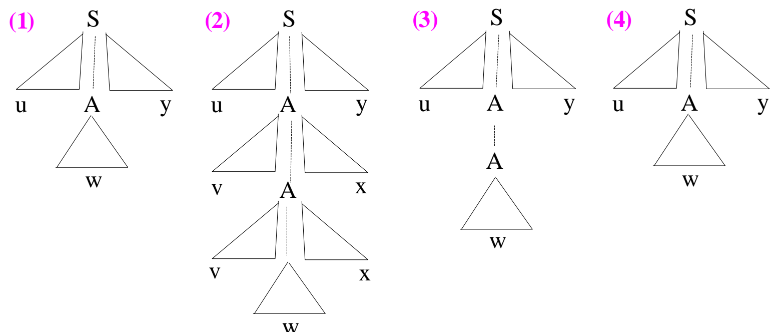
Let L be a CF language. Then *there exists* natural numbers p and q ≥ 0 depending **only** on L, such that if z ∈ L, and if |z| > p, then there is a *factorisation* z = uvwxy satisfying the following three conditions: (i) |vwx| ≤ q; (ii) *v and x are not **both** empty*; and (iii) for all i ≥ 0, we have $uv^iwx^iy \in L$ (the "pumping property").

Proof: Let G be a *CF grammar* for L. Let m be the length of the largest right hand side of any production in G. Let |N| = k. Put $p = m^k$. Let z ∈ L such that $|z| > p = m^k$. By *Lemma 1*, in any derivation tree for z, the **length** of some path must be ≥ k+1. (*If there is *more than one* derivation tree for z, choose the one with the **smallest** number of vertices*). But |N| = k, and so some *non-terminal* must be duplicated on this path.

Consider the **lowest possible duplicated pair** on this path. Call the duplicated non-terminal A. Then the *upper A* of this pair contains no other duplicates below it. This means that the length of the path from the upper A to the leaf is at **most** k+1 — because there are k non-terminals. Let us call the string *derived from the tree* rooted at the upper A 'vwx', where w is the string derived from the lower A. By Lemma 1, $|vwx| \le m^{k+1} = q$, say. We write z = u(vwx)y, as *shown on the right.*



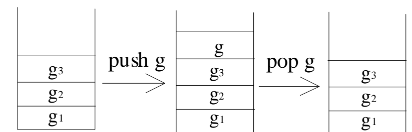
We now apply the subtree replacement principle. If we replace the upper A-tree by the lower A-tree, we *obtain* diagram (1). Thus $uwy = uv^0wx^0y \in L(G)$. If we replace the **lower** A-tree by the **upper** A-tree i times, we get $uv^{i+1}wx^{i+1}y \in L(G)$. Thus we get $uv^2wx^2y \in L(G)$ as shown in diagram (2).



We **conclude** the proof by showing that v and x cannot *both* be empty. Suppose to the **contrary** that v = x = ε. Then our *derivation tree* would be as shown in diagram (3). We could then **replace** the upper A-tree by the lower A-tree to obtain *diagram* (4). Thus we have shortened the dotted path by at least one, and this **contradicts** our choice of derivation tree of z made in (*) above. Hence v and x cannot both be empty. **End of proof**.

Application: $L = \{a^n b^n c^n : n \geq 0\}$ is not CF. **Proof**: *Suppose* that L were CF. Then we could find p and $q \geq 0$ s.t. for all $z \in L$ ($|z| > p$) the conditions (i), (ii), (iii) of the Pumping Lemma would be *satisfied*. Let $r > p,q$. **Consider** $z = a^r b^r c^r$. Then $z \in L$, and $|z| = 3r > p$. According to the *Pumping Lemma,* we can factorise z = uvwxy in **such** a way that (i) $|vwx| \leq q$; (ii) at most one of v and x is *empty*; and (iii) for all $i \geq 0$, $uv^i wx^i y \in L$.

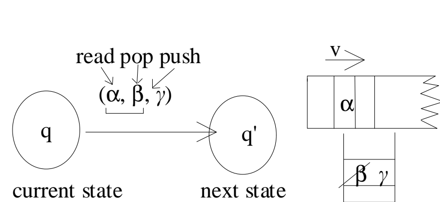
There are **5 cases** to consider: (1) vwx is *entirely* within the a-block; (2) .... the b-block; (3) .... the c-block; (4) vwx **falls across** the a|b boundary; (5) ....b|c.... <u>BUT</u>, vwx cannot fall across *both* the a|b and b|c boundaries — because $|vwx| \leq q < r$. By (iii), we **have** that $uv^2wx^2y \in L$. But if we now *consider* each of the 5 possibilities in turn: (1) **increased** the no. of **a's**, but the no. of *b's and c's* is still r; (2) ...**b's** ... *a's* ... *c's* ....; (3) ....**c's** ..... *a's* ... *b's*; (4)... **a's** and **b's** .... *c's* is still r; (5) ....**b's** and **c's** .... *a's* is still r. Thus $uv^2wx^2y \notin L$. Therefore, **<u>we have a contradiction</u>**, and it *follows* that L is not CF.



16th November 2000

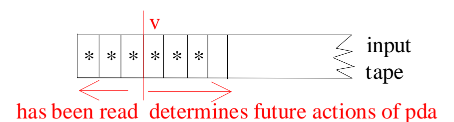
Pushdown Automata

<u>Idea</u>: A *pushdown automaton* (pda) is made up of a non-det ε-machine and a pushdown stack. A **<u>stack</u>** has a stack alphabet Γ where $g_i \in \Gamma$. We can only *read* the symbol at the top of the stack. Adding a symbol g to the **top** of the stack is called *pushing* g. The process of reading the symbol g at the **top** of the stack is called *popping* g.



<u>Definition</u>: A pushdown automaton (pda) is a 6-tuple <u>A</u>: <u>A</u> = (Q, Σ, Γ, δ, s, F), where Q is a *finite* set of states; Σ is an *input* alphabet; Γ is the stack alphabet; s is the *start* state; F is the set of *terminal* states; and δ is the **transition** function as shown on the left, where $\alpha \in \Sigma \cup \{\varepsilon\}$, and $\beta, \gamma \in \Gamma \cup \{\varepsilon\}$.

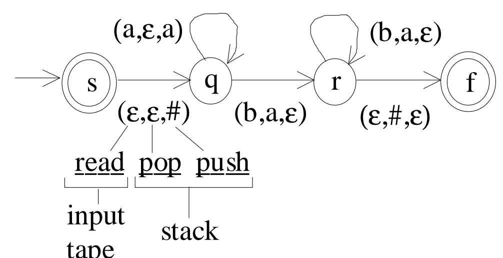
The operation of a pda is determined *non-deterministically* by **three** pieces of information: the *current symbol* at the top of stack, the current *state*, and the current *symbol* being read. The **<u>configuration</u>** of a pda is a triple (w, q, α), where $w \in \Gamma^*$ is the stack contents: (....w....)top; $q \in Q$ is the current state; and $\alpha \in \Sigma^*$ is the part of the input string under the tape head and to the **right**. Suppose that the *current configuration* is (ug, q, aα). Then a **possible** next configuration is (uh, q', α). For this, we write (ug, q, aα) → (uh, q', α), where → signifies a *move in the pda.*

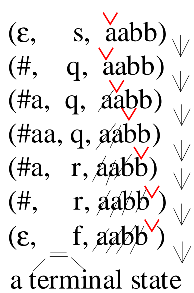


(a, g, h) q → q' ∈ δ

Example of a Pushdown Automaton

In the *diagram*, <u>A</u> = (Q, Σ, Γ, δ, s, F), *where* Q = {s, q, r, f}, Σ = {a,b}, Γ = {a, b, #}, F = {s, f}, and δ is as **shown** in the diagram.





How $\underline{A}$ processes the string a^2b^2. In the *diagram*, there is a ∨ over the letters on the tape being **read** — and *crossed out letters* indicate which **have** been read. In the notation of the definition of a "configuration", these *strings* would be written as *aabb, aabb, abb, bb, b,* ***e*** and ***e***. According to the diagram, a^2b^2 is **accepted** by $\underline{A}$. Clearly **all** strings of the form a^nb^n (where $n \geq 0$) are *accepted* by $\underline{A}$. Any string **beginning** with "b" will be rejected since the *transition* from q to r cannot be followed. If a string is to be accepted, then it **must** have the form $\alpha = a^k\beta$, where β does not *begin* with an "a".

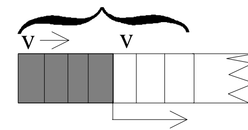
Let's see **how** α is processed. $(\varepsilon, s, a^k\beta) \rightarrow (\#, q, a^k\beta) \rightarrow (\#a, q, a^{k-1}\beta) \rightarrow (\#a^2, q, a^{k-2}\beta) \rightarrow \ldots \rightarrow (\#a^k, q, \beta)$. If $\beta = \varepsilon$, then the machine **halts** in state q. Thus we can **write** $\beta = b^m\gamma$, where $m \geq 1$, and γ does not *begin* with a "b". Therefore, we **have** $(\#a^k, q, b^m\gamma) \rightarrow (\#a^{k-1}, r, b^{m-1}\gamma) \rightarrow (\#a^{k-2}, r, b^{m-2}\gamma) \rightarrow \ldots$

If $m < k$, then we *reach* $(\#a^{k-m}, r, \gamma)$. If $\gamma = \varepsilon$, then the machine *terminates* in state r; if $\gamma \neq \varepsilon$, it **crashes**. Thus α is *not accepted.* If $m > k$, then we reach $(\#, r, b^{m-k}\gamma)$, and the machine **crashes**. If $m = k$, then we *reach* $(\#, r, \gamma)$. We know that γ can't begin with a "b". If it begins *with an* "a", then the machine **crashes**. Thus the only way α can reach the terminal state is *if* $\alpha = a^kb^k$. Conclusion: we have *proved that* $L(\underline{A}) = \{a^nb^n : n \geq 0\}$.

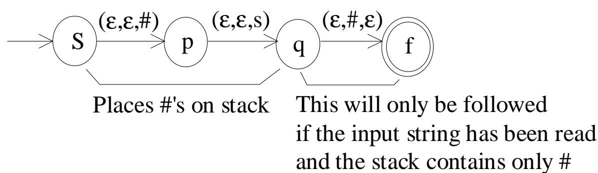
21st November 2000

Configuration of a pda

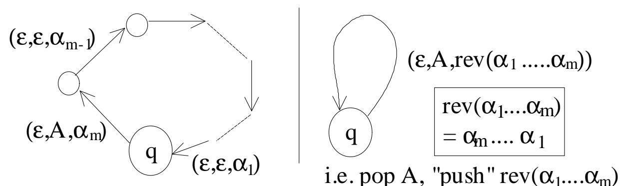
$w \in \Gamma^*$ *contents* of stack; $q \in Q$ *current* state; $\alpha \in \Sigma^*$ part of the *input* under the tape head and to the right. If c_1 and c_2 are two *configurations* linked by a **sequence** of moves, then we write $c_1 \text{—*→} c_2$. **Start** configuration: (ε, S, α) (stack *empty*, *start* state, *input* string). **End** config.: (γ, t, ε) (stack: *any* contents, *terminal* state, *input* string read). Let $\alpha \in \Sigma^*$. We say that α is accepted by $\underline{A}$ (**in final state mode**) if *there exists* $(\varepsilon, S, \alpha) \text{—*→} (\gamma, t, \varepsilon)$ for *some* $t \in F$. **Remember** that pda are *intrinsically non-det.* $L(\underline{A}) = \{\alpha \in \Sigma^* : \underline{A} \textit{ accepts } \alpha\}$.



Step 1



Step 2 For each production $A \rightarrow \alpha_1 \ldots \alpha_m \in P$



Step 3 For each input letter $a \in \Sigma$, add petal

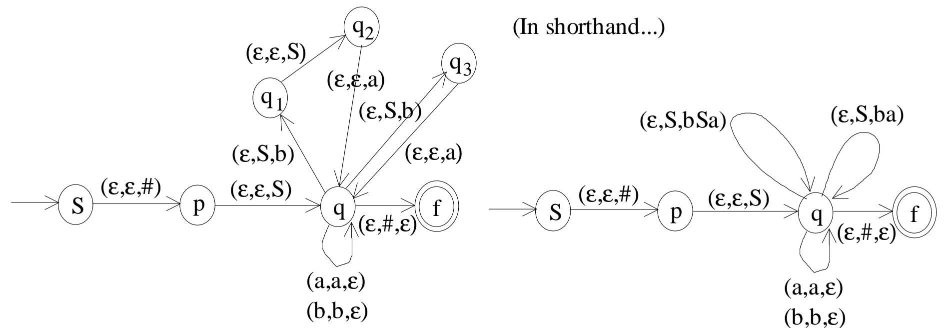
Step 4 Number States

We shall *prove* the following: if $\underline{G}$ is a CF grammar, there is a pda $\underline{A}$ such that $L(\underline{G}) = L(\underline{A})$. Let $\underline{G} = (N, \Sigma, P, S)$. The *procedure* for **constructing** $\underline{A}$ is as shown in steps 1 to 4 in the *diagram*.

We wish to **prove** that $L(\underline{A}) = L(\underline{G})$. *Recall that* $w \in L(\underline{G}) \Leftrightarrow S \Rightarrow^* w \Leftrightarrow S \Rightarrow^* w$ which is *left most.* The **machine** $\underline{A}$ will simulate a *left most derivation* of w. **Idea**: How to determine if a string $a_1 \ldots a_m \in \Sigma^*$ is generated by $\underline{G}$. We need only find a **left most** derivation of $a_1 \ldots a_m$. Suppose that we can *generate* $S \Rightarrow^* a_1 \ldots a_i\alpha$, with $\alpha \in (\Sigma+N)^*$. Then we need only *generate* $\alpha_{i+1} \ldots \alpha_m$ from α. The stack will contain **strings** used in *deriving* w from S, $A \rightarrow \alpha_1 \ldots \alpha_n$.

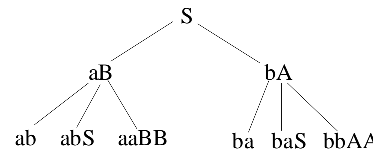
Example: From a CF Grammar to a pda

Let $\underline{G}$ be the grammar $S \to aSb \mid ab$. We follow the *method of the lectures* to obtain the following **shown** pda.



Assignment 3: Set 21/11; In 28/11; Back 5/12

Q: Let $\underline{G} = (N, \Sigma, P, S)$, *where* $N = \{S, A, B\}$, $\Sigma = \{a, b\}$, and P consists of the **following** productions: $S \to aB$, $S \to bA$, $A \to a$, $A \to aS$, $A \to bAA$, $B \to b$, $B \to bS$, and $B \to aBB$. *Describe* L($\underline{G}$). A: Consider the shown *tree* which may be used to **start** a derivation. Clearly, all strings in L($\underline{G}$) will have a length of at *least* 2. Further, all strings in L($\underline{G}$) will have **even** length. To see this, try to *derive* an odd string — it is impossible! (e.g. (babba): $S \to bA \to baS \to babA \to babbAA \to babbaA$ ×). So if $l \in$ L($\underline{G}$), we know that $|l| \geq 2$, and that $|l| = 2z$, $z \in \mathbf{Z}$.



Finally, *all* strings in L($\underline{G}$) will have to have the **same** number of a's as b's. To see this, consider all *possible* strings of length 2 or 4, and let us try to **derive** them. **<u>Length 2</u>** (4 not shown): aa: $S \to aB \to aaBB$ ×; ab: $S \to aB \to ab$ OK; ba: $S \to bA \to ba$ OK; bb: $A \to bA \to bbAA$ ×. *As you can see,* the only **valid** strings are those with # a's = # b's.

Therefore, L($\underline{G}$) = **all** strings where # a's = # b's. This implies that all *string lengths are even* — we cannot have an **odd** string where # a's = # b's. However, the above definition allows the case where # a's = # b's = 0 (the empty string), but the *productions* P do **not** allow this. To correct this, **alter** the definition to L($\underline{G}$) = all strings with an *equal* number of a's and b's, **excluding** the empty string.

Q: Use the **Pumping Lemma** to prove that the *language* $\{a^i b^i a^i : i \in \mathbf{N}\}$ is **not** CF. A: This proof has *already* been given in the lecture notes. All that is changed is where we had $\{a^i b^i c^i : i \geq 0\}$ before, we **now** have $\{a^i b^i a^i : i \in \mathbf{N}\}$. Otherwise the proof is **identical**.

Q: Let $L = \{a^i b^i a^j : i, j \in \mathbf{N}\}$. (i) Find a *CF grammar* for L. (ii) Let $L' = \{a^i b^j a^j : i, j \in \mathbf{N}\}$. Find a *CF grammar* for L'. (iii) **Describe** $L \cap L'$. Deduce that the *intersection* of CF languages need **not** be CF. A: (i) Let $\underline{G} = (N, \Sigma, P, S)$, where $N = \{S, A, B\}$, $\Sigma = \{a, b\}$, and P *consists of the following productions:* (1) $S \to aAbaB$; (2) $A \to aAb|\varepsilon$; and (3) $B \to aB|\varepsilon$. (Note: take **N** to be the *sequence* {1,2,3,...}. Do **<u>not</u>** do this in an *exam* — according to MVL, $0 \in \mathbf{N}$). (ii) Let $\underline{G} = (N, \Sigma, P, S)$, where $N = \{S, A, B\}$, $\Sigma = \{a, b\}$, and P *consists of the following productions*: (1) $S \to AabBa$, (2) $A \to Aa|\varepsilon$, and (3) $B \to bBa|e$.

(iii) In L', we have a <u>sequence of i a's followed by j b's</u>. In L, we have <u>i a's followed by i b's</u>. Clearly, if a string is in **both** L and L', we must have <u>i = j in L</u>'. This way, a string in L' which is also in L *starts* with i a's and then we **also** have i b's to *correspond* to what happens in L. Similarly, in L, we have (after a sequence of i a's) a <u>string of i b's and then j a's.</u>

In L’, we have (after a sequence of i a’s) a <u>string of j b’s and then j a’s</u>. Again, if a string is in **both** L and in L’, we must have <u>i = j in L</u>. This way, a string in L which is *also* in L’ starts with i a’s and then we have i b’s, **followed** by i a’s to correspond to what happens in L’. From the above, we *deduce* that for a string to be in both L and L’, i.e. in $L \cap L'$, we **must** have i = j in **both** L and L’. But if i = j, we can *say* that $L \cap L' = \{a^i b^i a^i : i \in \mathbf{N}\}$. But we have *already proved in question 2* that this language is **not** CF. Therefore, we have seen evidence that the <u>intersection</u> of two CF languages is not *necessarily* **itself** CF.

28th November 2000

We have <u>G</u>, a *CF grammar*, and <u>A</u>, a pda. We have an **informal** description of how <u>A</u> works. We will now <u>**prove**</u> that L(<u>G</u>) = L(<u>A</u>). <u>*Key Lemma*</u> (⊕): (#rev(σ), q, ρρ’) (the configuration of A, where ρ = input **already** read, and red will now denote *“already read”*) ⇔ $S \Rightarrow^* \rho\sigma$ in <u>G</u>. <u>**Proof:**</u> Once we have *initialised the stack*, we are in configuration (#S, q, w). On the **other** hand, $S \Rightarrow^* S$.

Thus (⊕) is true right at the *beginning* of processing w. Now assume that (⊕) is true at some stage — we prove that after a **transition** in <u>A</u>, it will still be true. <u>(Case 1):</u> Top of the stack contains a *non-teminal*. The **current** situation is (#rev(τ)A, q, ρρ’) ⇔ $S \Rightarrow^* \rho A \tau$. And $A \rightarrow \alpha_1 \ldots \alpha_m \in P$. Thus we have the *diagram* shown on the right.

q: $(\varepsilon, A, \alpha_m \ldots \alpha_1)$

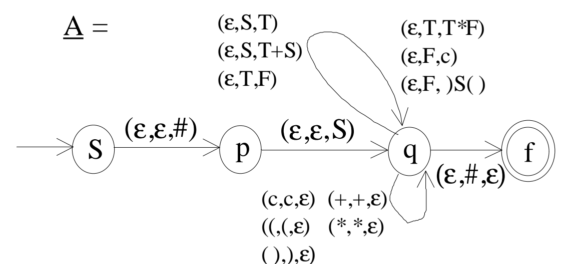
So in <u>A</u>, (#rev(τ)A, q, ρρ’) $\rightarrow^*$ (#rev(τ)$\alpha_m \ldots \alpha_1$, q, ρρ’). In <u>G</u>, $S \Rightarrow^* \rho A \tau$ (A = **left most** terminal) ⇒ $\rho\alpha_1 \ldots \alpha_m \tau$. For these expressions, (⊕) holds (**match** the blue expressions). <u>(Case 2)</u>: Top of the stack contains a *terminal* symbol. The **current** situation is (#rev(τ)x, q, ψxρ’) ⇔ $S \Rightarrow^* \psi x \tau$. So we have the *diagram* shown on the right. (x is a **terminal**).

q: (x,x,ε)

In <u>A</u>, (#rev(τ)x, q, ψxρ’) → (#rev(τ), q, ψxρ’). In <u>G</u>, $S \Rightarrow^* \psi x \tau$. *No change*, and (⊕) holds. **End** of proof. Now apply the <u>Key Lemma</u>: w ∈ L(<u>A</u>) ⇔ (ε,S,w) $\rightarrow^*$ (χ,f,ε) (*definition*) ⇔ (ε,s,w) $\rightarrow^*$ (ε,f,ε) (*in <u>A</u>*) ⇔ (#S, q, w) $\rightarrow^*$ (ε,f,ε) = (ε,f,wε) ⇔ (*key* lemma) $S \Rightarrow^* w\varepsilon = w$ ⇔ w ∈ L(<u>G</u>) (by *leftmost derivation*).

<u>Exercises</u>

Let <u>G</u> = (N, Σ, P, S), where N = {S, T, F}, Σ = {c, (,), +, *}, and P is *given* by S → T | S+T (1 and 2); T → F | F*T (3 and 4); and F → c | (S) (5 and 6). (1) Find a *left most derivation* for the string c*c+c. (2) **Construct** the pda <u>A</u> from <u>G</u> according to the *recipe* of the lectures. (3) Show the **configurations** involved in the *processing* by <u>A</u> of the string c*c+c.



A: (1) S →*2* S+T →*1* T+T →*4* F*T+T →*5* c*T+T →*3* c*F+T →*5* c*c+T →*3* c*c+F →*5* c*c+c. (2) The *diagram* on the right. (3) (ε, S, c*c+c) → (#, p, c*c+c) → (#S, q, c*c+c) →(2) (#T+S, q, c*c+c) →(1) (#T+T, q, c*c+c) →(4) (#T+T*F, q, c*c+c) →(5) (#T+T*c, q, c*c+c) → (#T+T*, q, *c+c) → (#T+T, q, c+c) →(3) (#T+F, q, c+c) →(5) (#T+c, q, c+c) → (#T+, q, +c) → (#T, q, c) →(3) (#F, q, c) →(5) (#c, q, c) → (#, q, ε) → (ε, f, ε). We are now at a *terminal state* ⇒ c*c+c is **accepted** by <u>A</u>.

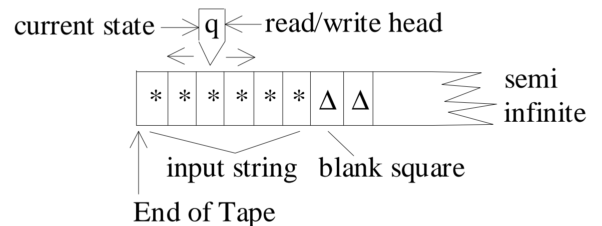
Concluding Remarks

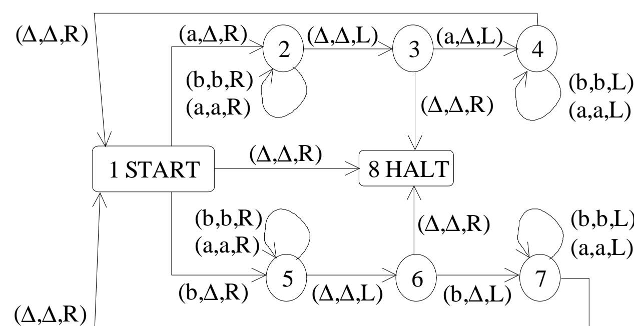
We have proved that *every CF language* can be recognised by a pda. **Theorem** (not proved here): If a language is *recognised* by a pda, then it **is** CF. If L and M *are* CF, then so too are L+M, L•M and L*. *However*, if L and M are CF, it is **NOT** true in general then L∩M is CF (look at a counter example). It *follows* that the **complement** of a CF language is not necessarily CF. One can define *deterministic* pda — they recognise *deterministic* CF languages. N.B.: Not **all** CF languages can be recognised by deterministic pda's.

3. Turing Machines

f.s.a. ⊆ pda ⊆ Turing Machines & regular languages ⊆ CF languages ⊆ r.e. languages. This is the *hierarchy* of languages known as the **Chomsky** Hierarchy. TM's were first defined in 1936 by Alan Turing. They were introduced to solve a problem in *Mathematical Logic*, and are now the basis of *Theoretical* Computer Science.

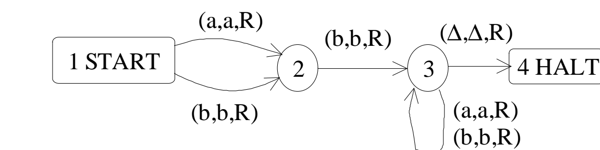
Informal Description of TM's: in the diagram. To increase the **power** of a f.s.a., (1) the tape head can move at any instant one square to the left or **right**; (2) the tape head can **overprint** the contents of a square.





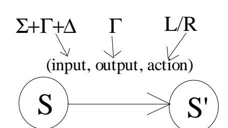
EXAMPLE 1

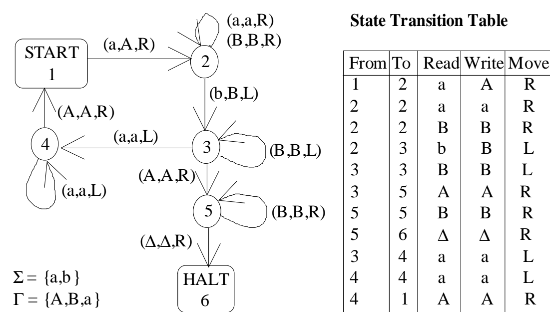
For *Example 1*, we can represent an **execution chain** by the diagram on the right, or by the following sequence: 1bab → Δ5ab → Δa5b → Δab5 → Δa6b → Δ7a → Δ7Δa → Δ1a → ΔΔ2Δ → Δ3Δ → ΔΔ8Δ. **Halt**!, and the string bab is thus *accepted* by this TM.



EXAMPLE 2 Σ = Γ = {a,b}

(1) Σ is the **input** string, and Δ ∉ Σ because we use Δ's to mark the *end* of an input string. (2) The tape head can **read** the contents of a square, **replace** the contents of a square, and **move** either L (left) or R (right). The tape-head can *never move L* of the first square. If it is told to do so, the TM crashes. (3) Γ is the set of symbols which can be *printed*. (4) A finite set of states, including one **start** state and some **halt** states which cause the TM to stop. (5) *Transitions*, examples of which are shown.





State Transition Table

From	To	Read	Write	Move
1	2	a	A	R
2	2	a	a	R
2	2	B	B	R
2	3	b	B	L
3	3	B	B	L
3	5	A	A	R
5	5	B	B	R
5	6	Δ	Δ	R
3	4	a	a	L
4	4	a	a	L
4	1	A	A	R

Example 3

Formal Definitions

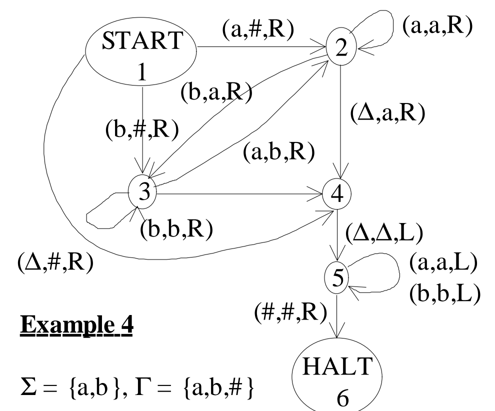
Definition: A *Turing Machine* is a 6-tuple $\underline{A} = (S, \Sigma, start, \Gamma, \delta, F)$. (Detail: S = *a finite set of states*, Σ = *the input alphabet*, start = *an initial state*, Γ = *the output alphabet*, δ = *the transition function*, and F = *a set of halt states*). Here, $\delta: S\times(\Gamma+\Sigma+\Delta) \to S\times\Gamma\times\{L,R\}$, and δ is a *partial function*, i.e. TM's can **crash**. An **instantaneous description** (ID) of A is a string $a_1...a_{i-1}na_i...a_r$, where $n \in S$; $a_1....a_{i-1} \in (\Gamma+\Sigma+\Delta)^*$; and $a_i...a_r \in (\Gamma+\Sigma+\Delta)^+$.

We can now **define** what we mean by a *computation* in A. Let the (ID) be $a_1...a_{i-1}na_i...a_r$, *and* let $\delta(n, a_i) = (n', a_i', R)$. Then we **write** $a_1...a_{i-1}na_i...a_r \to a_1...a_{i-1}a_i'na_{i+1}...a_r$ as "one move". Now **let** the (ID) be $a_1...a_{i-1}na_i...a_r$, and let $\delta(n, a_i) = (n', a_i', L)$. Then we **write** $a_1...a_{i-1}na_i...a_r \to a_i...na_{i-1}a_i'...a_r$ as another move.

Edge Effects. If the (ID) *is* $a_1...a_{i-1}na_i$, and we have $\delta(n, a_i) = (n', a_i', R)$, then we **write** $a_1...a_{i-1}na_i \to a_1...a_{i-1}a_i'n'\Delta$. If the (ID) is $na_1...a_i$, and if $\delta(n, a_i) = (n', a_i', L)$, then the machine **crashes**. If α and β are ID's, and if β is *obtained* from α by a sequence of **moves**, then we write $\alpha \to^* \beta$. **Definition**: If A is a TM, the language *accepted* by A, written L(A), is defined by $L(\underline{A}) = \{w \in \Sigma^*: \text{start } w \to^* u \text{ halt } v$, *where* u and $v \in (\Gamma+\Sigma+\Delta)^*\}$.

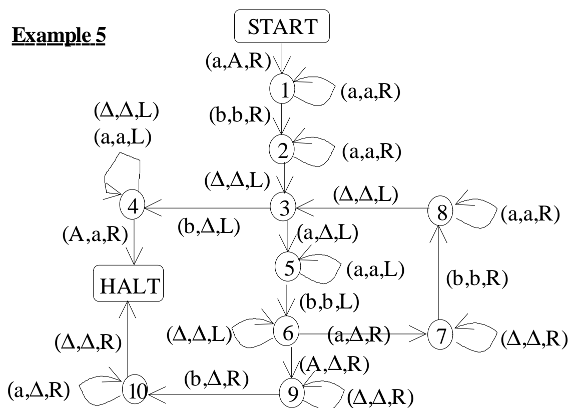
Theorem (not *proved* here): A language is accepted by a TM ⇔ it is *recursively enumerable*. So far, our automata has only been used for **accepting** languages. We now want to consider TM's as *transducers*, i.e. we are interested in the string **left on the tape** when the TM halts.

Example 4. Q: *Process* aba. A: 1aba → #2ba → #a3a → #ab2 → #aba4 → #ab5a → #a5ba → #5aba → 5#aba → #6aba. *HALT*. #aba is the string **left** on the tape when the machine halts. This TM is computing a *function,* with input any string in (a+b)*, e.g. w; and output #w. We can also compute functions concerning **N**. The *ordered* m-tuple $(x_1,...,x_m) \in \mathbf{N}^m$ will be *represented* by the string $a^{x_1}ba^{x_2}b...ba^{x_m}$.



Examples: (i) abaa = $a^1ba^2 \leftrightarrow (1,2) \in \mathbf{N}^2$; (ii) bbabbaa = $a^0ba^0ba^1ba^0ba^2 \leftrightarrow (0,0,1,0,2) \in \mathbf{N}^5$. We are interested in TM's which *accept as input* elements of $\mathbf{N}^m$ encoded as above, and when they halt (**if** they do) leave the *unary encoding* of a sequence from $\mathbf{N}^n$. A (*partial*) function $f: \mathbf{N}^m \to \mathbf{N}^n$ is said to be *Turing computable* if the following conditions are satisfied:

Let $D \subseteq \mathbf{N}^m$ on which f is defined. Then, if there *is* a TM $\underline{A}_f$, (1) $(x_1,...,x_m) \in D \leftrightarrow a^{x_1}b...ba^{x_m} \in L(\underline{A}_f)$; (2) If $(x_1,...,x_m) \in D$, then $a^{x_1}b...ba^{x_m} \to^* a^{y_1}b...ba^{y_n}$ (*halt*), where $f(x_1,...,x_m) = (y_1,...,y_n)$. Functions which can be computed *in this way* are called partial recursive functions.



Consider f: **N**\{0}×**N** → **N**, where f(m,n) = m-n *if* m > n, and f(m,n) = 0 *if* m ≤ n. **<u>Claim</u>**: Example 5 *computes* f. Now $a^+ba^* \ni a^mba^n$, where m ≥ 1, and $a^+ba^* \rightarrow^* a^{f(m,n)}$.

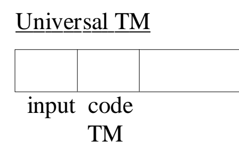
5th December 2000

<u>Turing's Thesis</u>

To say that something is *computable* means that we have an **algorithm** for computing it. But what *is* an "algorithm"? Dictionary definitions such as *"A step-by-step procedure by which an operation can be carried out without any exercise of intelligence"* are not correct in the **mathematical** sense. Can we do **better**?

Computability in Computer Science ⇔ **Mathematical Algorithm**. Example: *<u>Euclid's Algorithm</u>*, with input a, b ∈ **N**, and output gcd(a,b). We should agree that anything a TM does should be regarded as *algorithmic*. Turing's Thesis asserts that the converse is true — for each algorithm, there is a Turing Machine that **implements** the algorithm.

In other words, Turing's Thesis says that we define an *algorithmic* problem as one that can be solved by a Turing Machine. It implies that whatever a **computer** can do, however *powerful and sophisticated*, it can also be carried out by some Turing Machine, because computers do no more than implement algorithms. Thus the boundaries of the computable and the non-computable are defined by the abilities of **Turing Machines**.



<u>Encoding TM's</u>

Let Σ = {a,b}. **<u>Theorem</u>** (not proved): Let L be a r.e. Σ-language, then there *is a TM <u>A</u>* s.t. L(<u>A</u>) = L, with Σ = {a,b}, and Γ = {a,b,#}. We also *assume* that (i) the unique **start** state is labelled 1; and (ii) there is a unique **halt** state labelled 2 (easy to achieve). We'll call such Turing Machines "*Special*".

From	To	Read	Write	Move
x_1	x_2	x_3	x_4	x_5

x_3 x_4	code
a	aa
b	ab
Δ	ba
#	bb

x_5	code
L	a
R	b

We shall encode *each transition table* of a special TM by means of a **string** from {a,b}*. Encode a **row**: x_1, x_2 is encoded as $a^{x_1}ba^{x_2}b$. x_3, x_4 and x_5 are encoded as shown in the *diagram on the left*. We can encode every row of the **transition table** by means of a string from $a^+ba^+b(a+b)^5$ (*remember that* $^+$ = "*dagger*"). To encode the **whole** table, simply *concatenate the codes* for each of the rows.

Let $\mathcal{T}$ be the set of *all transition tables* of special TM's. Then <u>code</u>: $\mathcal{T}$→{a,b}* is such that <u>code</u>(transition table) = *concatenation of the row codes*. **Given** a string w ∈ {a,b}*, we can *easily decide* if there is a **transition table** <u>A</u> s.t. <u>code</u>(<u>A</u>) = w.

Example 1

String: *abaaabaaaabaaabaaabaaaabaaabaabababa*

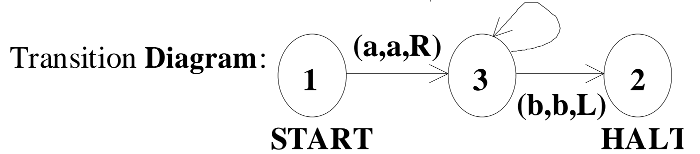
From	To	Read	Write	Move	Code for Each Row
1	1	b	b	R	abababb
1	3	a	b	R	abaaabaaabb
3	3	a	b	L	aaabaaabaaaba
3	2	Δ	b	L	aaabaabbaaba

The one-word code word for the whole machine is: *ababababbabaaabaaabbaaabaaabaaabaaaabaabbaaba*

Transition Table:

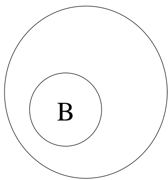
From	To	Read	Write	Move
1	3	a	a	R
3	3	a	a	R
3	2	b	b	L

Transition **Diagram**:



7th December 2000

Let Σ = {a,b}. **Define** G ⊆ Σ* as follows: G = {w ∈ im(<u>code</u>): if <u>code</u>(<u>A</u>) = w then w ∈ L(<u>A</u>)}. (*<u>Note</u>*: w ∈ im(<u>code</u>) is *interpreted as* w = <u>code</u>(<u>A</u>) for some <u>A</u> ∈ $\mathcal{T}$). Put B = Σ*\G. Is B r.e.? Suppose for the sake or *argument* that B *is* r.e. This simply means that B is *accepted by some TM*. Thus (by an argument given **earlier**) B is accepted by a **special TM**, <u>A</u>. Therefore, L(<u>A</u>) = B. Put w = <u>code</u>(<u>A</u>). We *either have* (1) w ∈ B, or (2) w ∉ B (but **NOT** both). **<u>Case 1:</u>** Suppose that w ∈ B, then w ∉ G. *Now* w = <u>code</u>(A), and thus w ∈ im(<u>code</u>). The *only* way that w ∉ G can occur is if w ∉ L(<u>A</u>) = B. *Contradiction* — case 1 cannot hold. **<u>Case 2:</u>** w ∉ B, and so w ∈ G. This means that w ∈ L(<u>A</u>). But L(<u>A</u>) = B, and **hence** w ∈ B. <u>Conclusion:</u> B is **<u>not</u>** r.e., and so by Turing's Thesis, no **algorithm** (and therefore no *program*) can determine <u>membership</u> of this language.

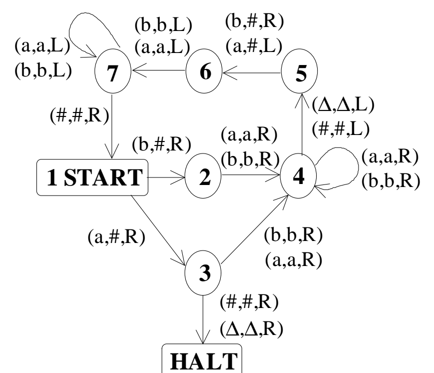


<u>Assignment 4: Set 5/12; In 12/2; Back 15/12</u>

Q: With reference to the transducer in *<u>Example 5</u>*, write down the **execution chain** corresponding to the input string a^3ba. A: Startaaaba → A1aaba → Aa1aba → Aaa1ba → Aaab2a → Aaaba2 → Aaab3aΔ → Aaa5bΔΔ → Aa6abΔΔ → AaΔ7bΔΔ → AaΔb8ΔΔ → AaΔ3bΔΔ → Aa4ΔΔΔΔ → A4aΔΔΔΔ → 4AaΔΔΔΔ →aHALTaΔΔΔΔ = aHALTa.

Because we are now at the *terminal* state, this means that the string a^3ba is **accepted** by the Turing Machine. What is left on the tape is the *string* aa = a^2. We would expect this, as example 5 represents the **function** f: **N**\{0}×**N**→**N**, where f(m,n) = m-n if m > n, and f(m,n) = 0 if m ≤ n. With an *input string* a^3ba ↔ (3,1) ∈ $\mathbf{N}^2$, we would *expect* the output 3-1 = 2 ↔ a^2. ✓

Q: Consider the TM shown. (i) Find the *execution chains* for (a) aaa, (b) aba, and (c) ababb. (ii) The **language** accepted by this TM is the set of all strings over {a,b} with an *odd number of letters* having an "a" in the middle. Explain the algorithm which this TM **implements**.



A: (i) 1aaa → #3aa → #a4a → #aa4 → #a5aΔ → #6a#Δ → 7#a#Δ → #1a#Δ → ##3#Δ → ###HALTΔ = ###HALT. *Therefore* aaa is **accepted** by the TM as we have reached the <u>terminal state</u>.

(b) 1aba → #3ba → #b4a → #ba4 → #b5aΔ → #6b#Δ → 7#b#Δ → #1b#Δ → ##2#Δ → the machine *crashes* (there is no transition reading a '#' at node **2**). *Therefore*, aba is NOT accepted by the TM — it *crashes* while processing the string.

(c) 1ababb → #3babb → #b4abb → #ba4bb → #bab4b → #babb4 → #bab5bΔ → #ba6b#Δ → #b7ab#Δ → #7bab#Δ → 7#bab#Δ → #1bab#Δ → ##2ab#Δ → ##a4b#Δ → ##ab4#Δ → ##a5b#Δ → ##6a##Δ → #7#a##Δ → ##1a##Δ → ###3##Δ → ####HALT#Δ = ####HALT#. *Therefore*, ababb is **accepted** by the TM — we have reached the terminal state.

(ii) The TM does not accept *even length strings* (aa, abab, ababbab, etc.) — it only accepts odd length strings with an 'a' in the **middle** (a, aab, bbaba, etc.). We can think of the TM in question as working in cycles, where in *each* cycle we replace 2 letters by a '#'. The letters we replace are the **first** and **last** letters of *whatever string of a's and b's* we have at a particular time.

When the length of the string of a's and b's is ≤ 1 after a *particular cycle*, we stop cycling and look at what we have left **apart** from the #'s. If we are left with an 'a', then we reach the *HALT* state. Otherwise (*we have 'b' or the empty string*) the machine crashes. In **pseudo-code**, the *algorithm* which the TM implements could be written as follows:

```
Let the input string to be processed be I.
Repeat
     remove the first letter from I
     remove the last letter from I
Until |I| ≤ 1
If I = a accept I
Else reject I.
```

Exam Paper: January 2001

SECTION 1 (Compulsory)

(1) (a) Let Σ be a fixed alphabet.
Define what is meant by a *recognisable language* over Σ. **[1 mark]**
Define what is meant by a *regular expression* over Σ. **[2 marks]**
State *Kleene's Theorem* without proof. **[1 mark]**

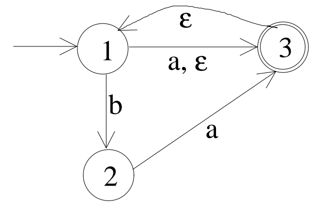
(b) Explain the algorithm for constructing an ε-automaton from a regular expression with reference to the regular expression (ab+a)*. **[6 marks]**

(c) Find a context-free grammar for the language $\{a^i b^j c^j a^i : i, j \geq 0\}$. Construct a leftmost derivation of the string ab^2c^2a and a corresponding derivation tree. **[5 marks]**

(d) Give the formal definition of a Turing machine. **[2 marks]**
Explain briefly what is meant by Turing's Thesis, you should make clear the relationship between 'algorithms' and Turing machines, and between Turing machines and real computers. **[3 marks]**

SECTION 2 (Answer 2 out of 4 questions)

(2) This question refers to the ε-automaton **A** in the shown diagram.



(a) Convert **A** into a non-deterministic machine withou ε-transitions **B** which recognises the same language as **A**. **[6 marks]**

(b) Convert **B** into a connected deterministic automaton **C** which recognises the same language as **B**. **[3 marks]**

(c) Write down the language equations associated with **C**. **[3 marks]**

(d) Solve the language equations in (c), and so find a regular expression for L(**A**). **[3 marks]**

(3) What is meant by a right-linear grammar? Prove that a language is generated by a right-linear grammar if and only if it is recognisable. **[15 marks]**

(4) This question concerns the context-free grammar **G** = (**N**, Σ, **P**, *S*) where **N** = {*S*, A}, Σ = {a,b} and **P** consists of the following productions:

1. $S \rightarrow AaA$
2. $A \rightarrow aA$
3. $A \rightarrow bA$
4. $A \rightarrow \varepsilon$

(a) Construct a leftmost derivation of the string a^3. **[2 marks]**

(b) Construct a pushdown automaton which recognises the language L(**G**). **[8 marks]**

(c) Show how your machine processes the string a^3 by tracing through the configurations assumed by your machine. **[5 marks]**

(5) A Turing machine is said to be *special* if it satisfies the following conditions:

$\Sigma = \{a, b\}$ and $\Gamma = \{a, b, \#\}$
There is a unique start-state labelled 1, and a unique halt-state labelled 2.

It is a theorem that every recursively enumerable Σ-language can be recognised by a special Turing machine.

(a) Explain how every special Turing machine transition table can be encoded by means of a string from (a+b)*. **[5 marks]**

(b) Let $G \subseteq (a+b)^*$ be the set of all these strings w which encode a Turing machine transition table **A** and satisfy $w \in$ L(**A**). Prove that the complement of G is not recursively enumerable. **[10 marks]**

(Questions done: 1, 2, 4)