Langages formels, calculabilité, complexité - L3S1

Regular expressions

`[A-Z][a-z]*[ ][A-Z][A-Z]` is an example of regexp.

Formal definition

1. $\varepsilon, \emptyset$ are regular expressions.
2. $\forall x \in \Sigma$, x is a regexp.
3. If R_1 and R_2 are two regexps, $R_1 + R_2$, $R_1 R_2$, R_1^* are regexps.

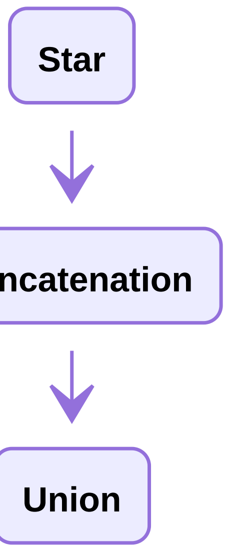
Language of a regexp

- $L(a) = \{a\}$, for $a \in \Sigma$
- $L(\varepsilon) = \{\varepsilon\}$
- $L(\emptyset) = \emptyset$
- $L(R_1 + R_2) = L(R_1) \cup L(R_2)$
- $L(R_1 R_2) = \{xy, (x, y) \in R_1 \times R_2\}$
- $L(R_1^*) = L(R_1)^*$

Examples

- $\{0 + 1\}^* 1$: any string that ends with a 1.
- Strings with alternating 0 and 1 : $(\varepsilon + 1)(01)^*(\varepsilon + 0)$

Order of operations



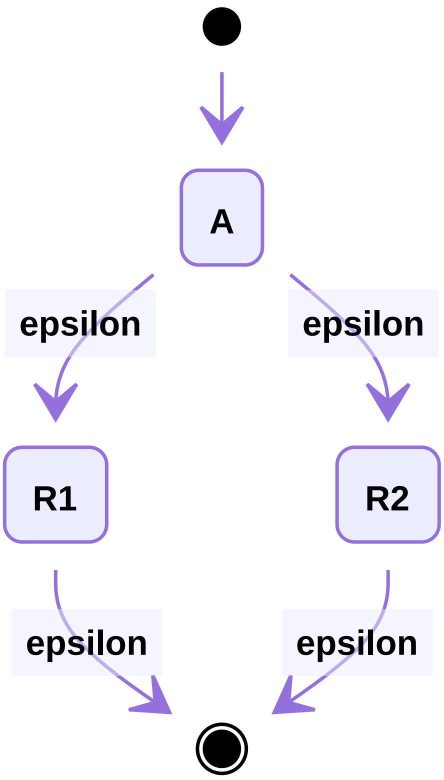
Equivalence with DFAs

Theorem : $L = L(R)$ for some regexp $R \Leftrightarrow L = L(D)$ for some DFA D.

Dem :

$L = L(R) \Rightarrow L = L(D)$

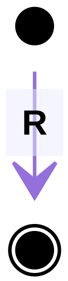
Idea: do it recursively.
It is trivial to do the base cases.
Union : with ε-transitions.



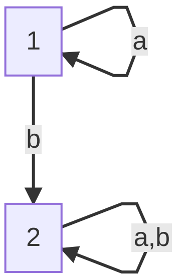
Concatenation : with ε-transitions.

Converting a DFA in a regexp :

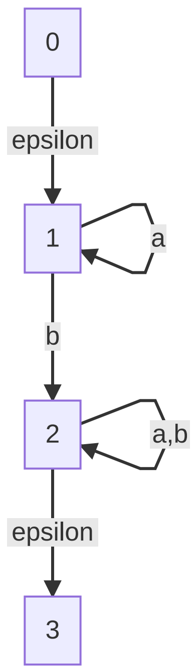
1. Assume only one q_{start} and one q_{accept}, $q_{start} \neq q_{accept}$.
2. No incoming transition to q_{start}
3. No out-coming transition from q_{accept}
4. Initially, k states, other than q_{start} and q_{accept}.
5. For $i = 1, \ldots, k$, remove node i
6. In the end :



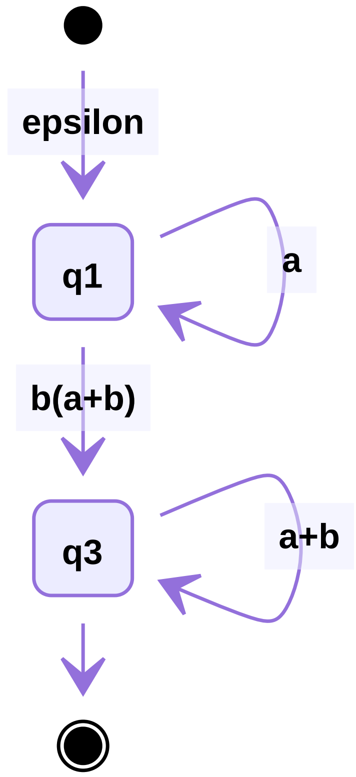
Example :



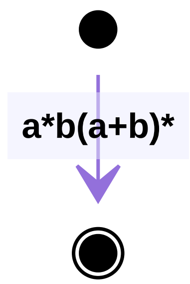
Then



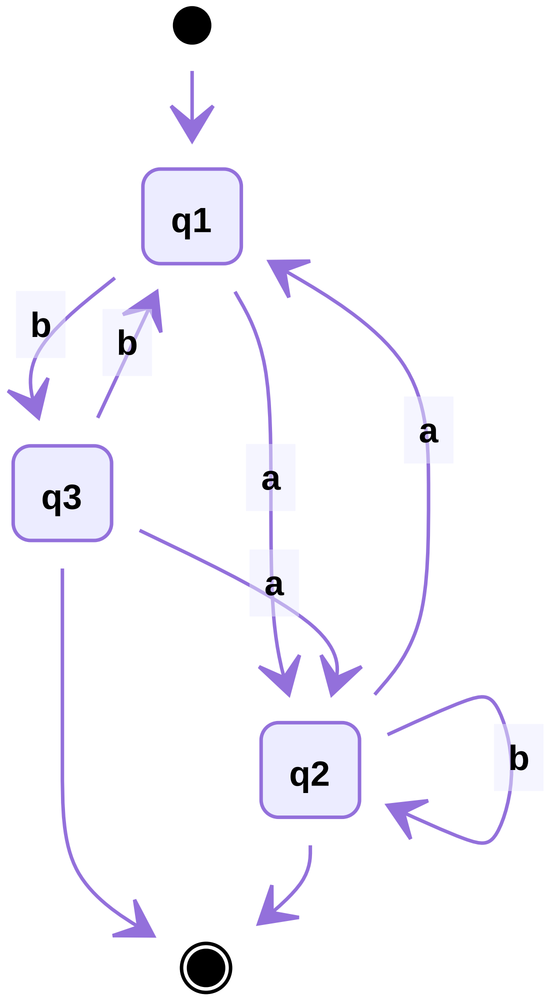
Then : (rip 2)



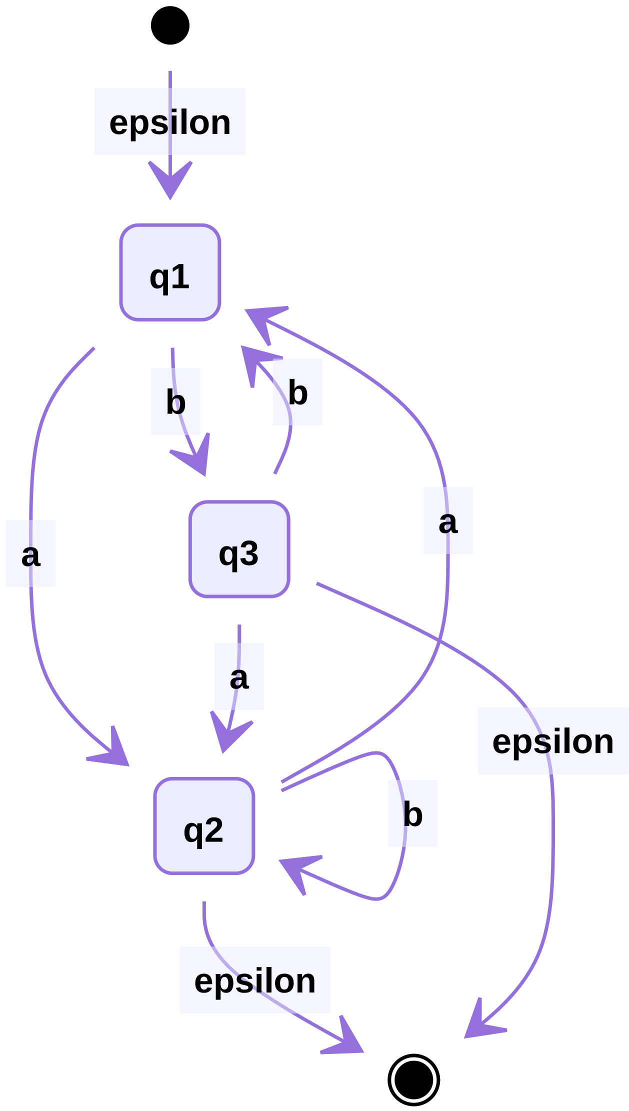
Finally, (rip 1)



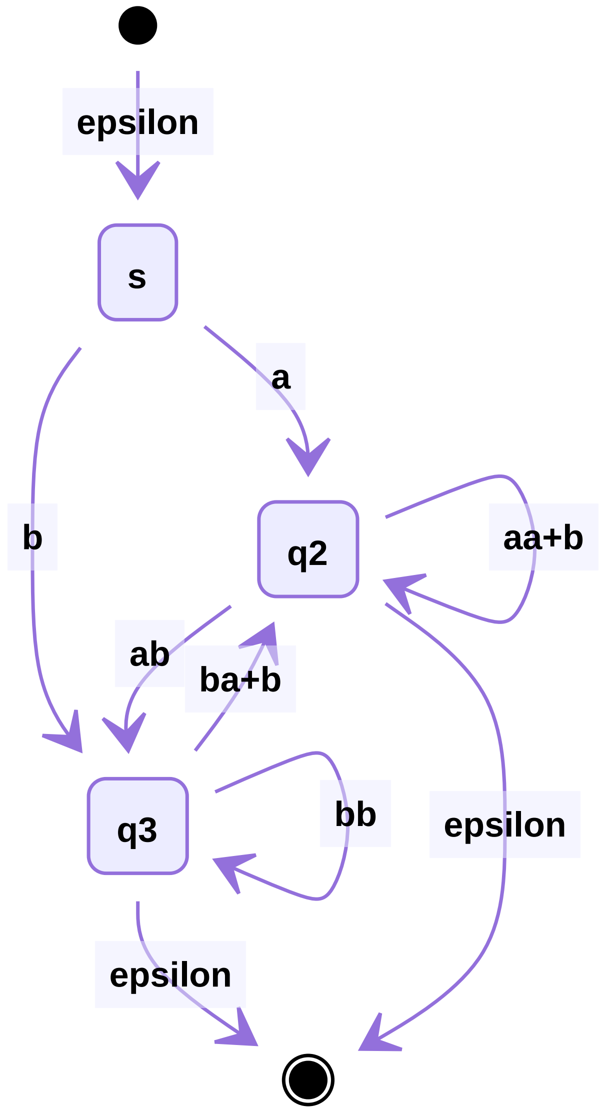
Another example : (accepting : 2 and 3)



Then :



Rip 1 :



And so on and so forth...

Proprieties of regexps

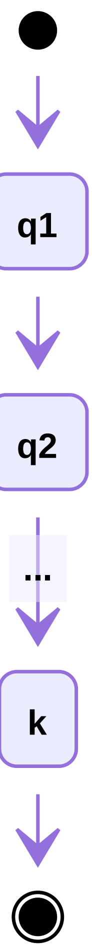
$L+M=M+L$, $(L+M)+N=L+(M+N)$, $(LM)N=L(MN)$, $\emptyset+L=L+\emptyset=L$, $\varepsilon L=L\varepsilon=L$, $\emptyset L=L\emptyset=\emptyset$, $\emptyset^*=\varepsilon$, $\varepsilon^*=\varepsilon$, $L(M+N)=LM+LN$, $(M+N)L=ML+NL$, $L+L=L$, $(L^*))^*=L^*$, $L^+=LL^*=L^*L$.

Pumping lemma

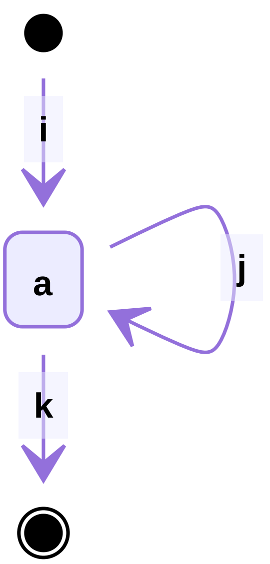
Is $L_{01}=\{0^n1^n, n\in\mathbb{N}\}$ regular ?

Suppose it is. It exists a DFA that recognises it with a finite number of states k such that $L(D)=L_{01}$.

What about 0^k1^k ?



After reading $k-1$ elements, we have exhausted the different states. After the k^{th} element, a state will be repeated. Therefore, if we read k 0s, the repeated state, a, will be reached.



We then know that $j \neq 0$. Then if $0^k 1^k$ is accepted, then $0^{k-j}1^k$ should also be accepted. Contradiction.

Pumping lemma : let $L = L(D)$ for a DFA D. Then $\exists p \in \mathbb{N}$ (the pumping length) such that $\forall w \in L$ with $|w| \geq p$, we can break w into $w = xyz$ such that :

1. $y \neq \varepsilon$ (non trivial)

2. $|xy| \leq p$ (xy "near the beginning")
3. $\forall i \geq 0, \quad xy^i z \in L$ (w can be "pumped")

Proof:

Let $p = |Q|$, $D = (Q, \Sigma, \delta, q_0, F)$. Let $w = a_1 \dots a_m$ with $m \geq p$ (if no such w exists, the proof is done).

Let $q_i := \hat{\delta}(q_0, a_1 \dots a_i) \quad i = 0, \dots, m$ (at least $p + 1$ of these).

By pigeonhole : $\exists i \neq j \leq p$ such that $q_i = q_j$.

Let $x := a_1 \dots a_i$; $y = a_{i+1} \dots a_j$; $z = a_{j+1} \dots a_m, \Rightarrow |xy| \leq p$.

x takes us from q_0 to q_i, y from q_i tp $q_j = q_i$, z from q_j to q_m, an accept state. (x or z may be ε, but $y \neq \varepsilon$ since $i < j$). Hence $xz \in L$ and $xy^k z \in L$ for any $k > 0$.

Examples

$L_{01} = \{0^n 1^n, n \geq 0\}$ is not regular. Suppose it is. Let p be the pumping length. Let $w = 0^p 1^p$. Not $|w| \geq p$.

Then $\exists x, y, z$ such that $w = xyz$, $|xy| \leq p$, $y \neq \varepsilon$ and $xy^i z \in L$ for $i \geq 0$

Since $|xy| \leq p$, y only has 0s. All the ones are in z. But $xz \in L$ by pumping lemma. Absurd.

Let $L = \{1^{n^2}, n \geq 0\}$.

Let $s := 1^{p^2}$. $y = 1^i$, with $0 < i \leq p$.

Then $\forall i \in \mathbb{N}, p^2 + i$ is a perfect square. Absurd.

Minimisation

Given a DFA D, does D have redundant states ? How can we find a minimal DFA that is equivalent to D ?

Definition: we say that two states p, q are equivalent iff for all $w \in \Sigma^*$, $\hat{\delta}(p, w) \in F \Leftrightarrow \hat{\delta}(q, w) \in F$. We write : $p \equiv q$.

Note: $\equiv$ is an equivalence relation : it partitions the states into equivalence classes.

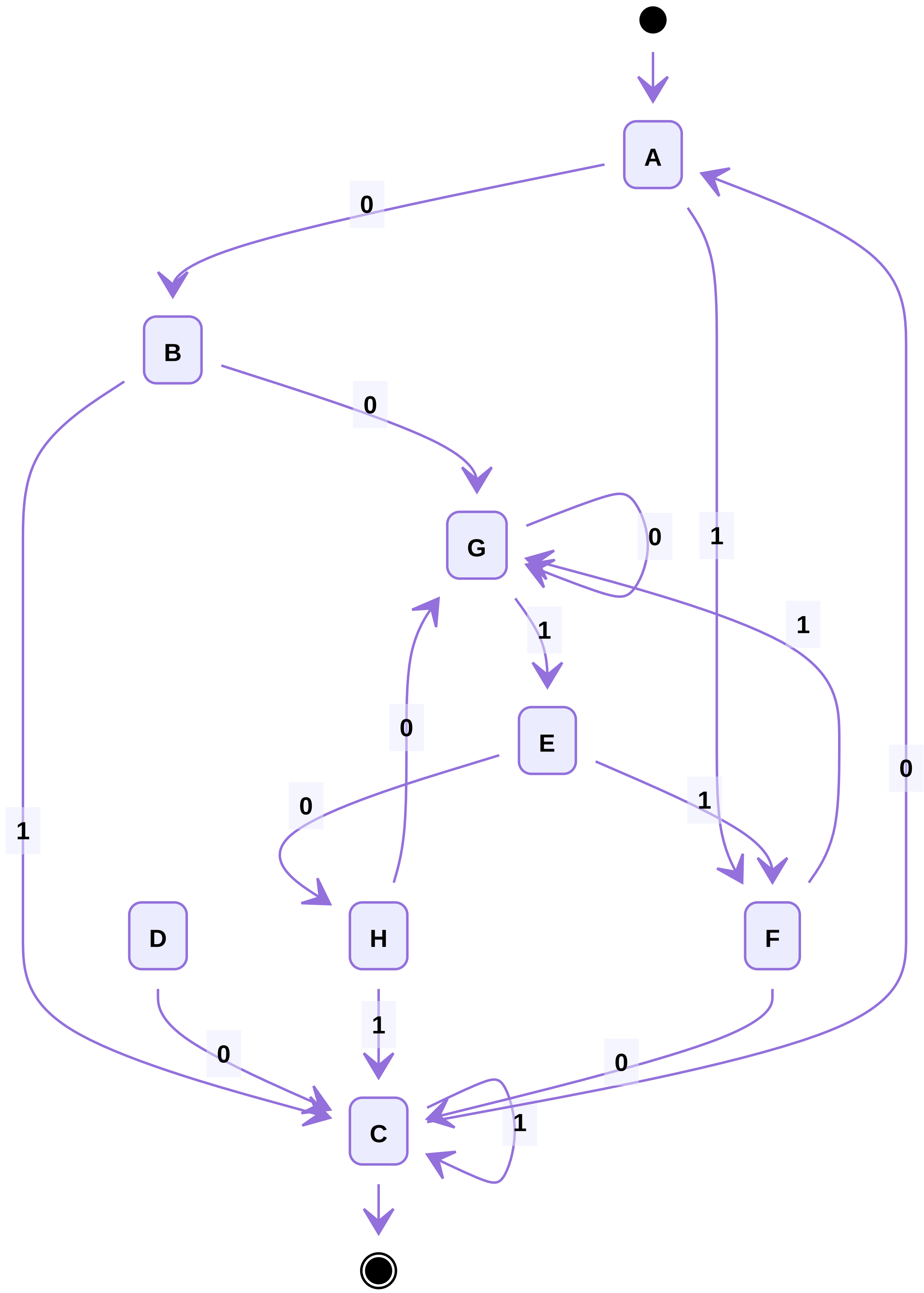
Note: if $p \equiv q$, $p \in F \Leftrightarrow q \in F$.

The equivalent states can be "merged into one".

Table filling algorithm

Find all distinguishable pairs : $p \not\equiv q \Rightarrow \exists w \in \Sigma^* \, \hat{\delta}(p, w) \in F \land \hat{\delta}(q, w) \notin F$, remove them recursively, conclude.

Idea: if for all $a \in \Sigma$, $\delta(r, a) = p, \delta(s, a) = q \ \land p \not\equiv q \Rightarrow r \not\equiv s$



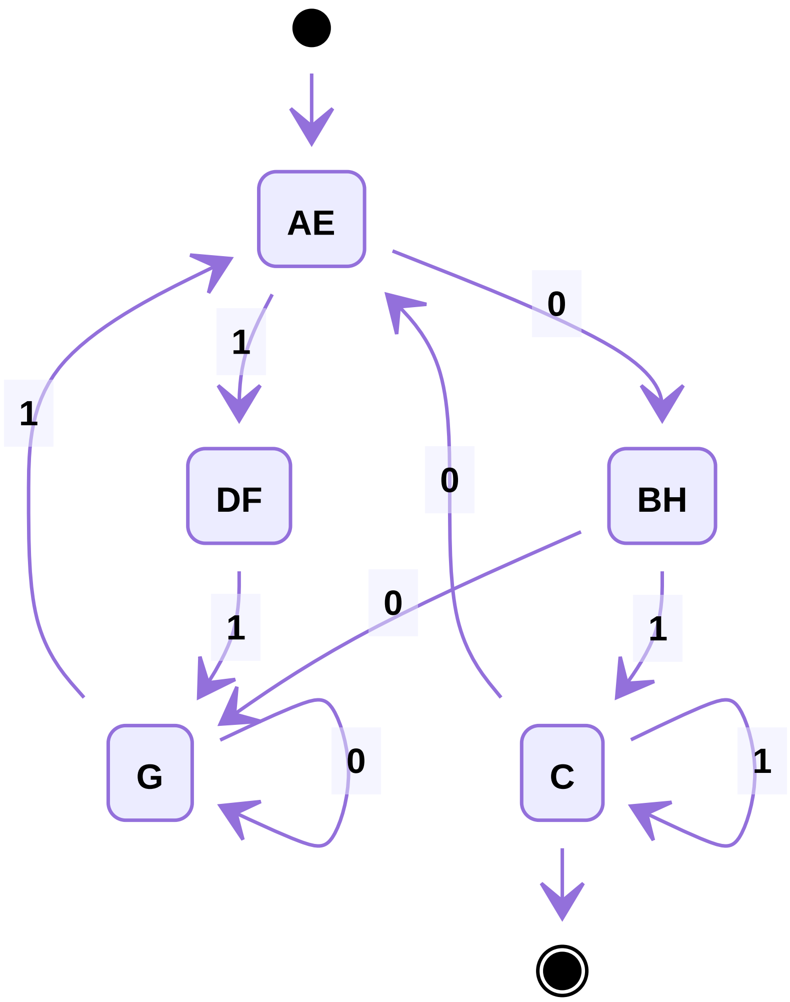
Distinguishable pairs table (partly filled)

x = distinct	A	B	C	D	E	F	G
A							
B							
C	x	x					
D			x				
E			x				
F			x		x		
G			x			x	
H			x				

Ex : $\{C, H\}$ pair, + 0 input $\Rightarrow \{G, F\}$ are distinguishable, +1 input : $\{E, F\}$ distinguishable.

We would see that $\{A, E\}, \{D, F\}, \{B, H\}$ are equivalent. By fusing the nodes, we can come with a smaller DFA that recognises the same language.

Final DFA :



Theorem: if two states are not distinguished by the table filling algorithm, they are equivalent.

Proof: Call $\{p, q\}$ the bad pair. We consider the pair with the shortest $\omega = a_1 \ldots a_n$.

$\hat{\delta}(p, \omega) \in F$, $\hat{\delta}(q, \omega) \notin F$. Let $r = \delta(p, a_1)$, $s = \delta(q, a_1)$. Therefore, r and s are distinguishable (otherwise, p and q wouldn't be). This contradicts the hypothesis of shortest ω. So $\{r, s\}$ is not a bad pair, so the algorithm finds this pair, and thus $\{p, q\}$ at the next step.

Context-free grammars

$L = \{0^N 1^N, N \geq 0\}$ is not a regular language. But it can be described recursively : $\varepsilon \in L$ and $\forall A \in L, 0A1 \in L$.

We note $\begin{matrix} A \to \varepsilon \\ A \to 0A1 \end{matrix}$, or $A \to 0A1|\varepsilon$. Another example is : $\begin{matrix} A \to a \\ A \to A + A|A * A \end{matrix}$.This defines a **context-free grammar**.

Formally, a **context-free grammar (CFG)** is a 4-tuple $G = (V, \Sigma, R, S)$, with V the set of variables, Σ the set of terminals (alphabet), R the set of rules/deductions, which are $A \to \omega$, with $\omega \in \{V \cup \Sigma\}^*$, $A \in V$ and S the start variable.

Definition : language recognised by a CFG

Let $A \to \omega$ be a rule, uAv yields $u\omega v$. More generally, u derives v ($u \xrightarrow{*} v$) if $u = v$ or $u \to u_1 \to \cdots \to u_k \to v$.

We define $L(G) = \{\omega \in \Sigma^*, s \xrightarrow{*} \omega\}$ the language recognised by the CFG G.

Example : Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA. Then there exists a CFG G that recognises $L(D)$.

If the transition function is of the form $\delta(q_i, a) = q_j$, we put $R_i \to aR_j$, with R_0 the start variable.

$R_0 \to aR_1 \to abR_2 \to ab$
$R_1 \to \varepsilon \text{ if } q_i \in F$

This CFG recognises the same language as the automaton.

Therefore, we've proven that CFGs recognise strictly more languages than DFAs.

Derivation, parse trees

Leftmost derivations

Idea : in each step, we will replace the *leftmost* variable.

Example : $S \to SS \mid (S) \mid \varepsilon$
For instance, we want to generate `(())()` :

$S \to SS \to (S)S \to ((S))S \to ((\varepsilon))S \to (())(S) \to (())(\varepsilon) = (())()$.

Parse trees

```
     S
   /   \
  S     S
 /|\   /|\
( S )( S )
  |     |
( S )   ε
  |
  ε
```

Sometimes, there is not uniqueness of the parse tree. We call such grammars **ambiguous** (ex : $S \to S + S \mid S * S \mid a$)

Ambiguity

Definition: A CFG is ambiguous if $\exists \omega \in L(G)$ with two or more parse trees.

We can remove ambiguity : $\begin{array}{l} B \to (RB \mid \varepsilon \\ R \to) \mid (RR \end{array}$ recognises the same language of parenthesis as before, but is unambiguous.

We say that L is **inherently ambiguous** if there is no unambiguous G such that $L(G) = L$.

An example of such a language would be : $L = \{0^i 1^j 2^p | i = j \vee j = k\}$. The ambiguity comes from $i = j = k$ (which is not context-free, we'll see that later on).

1. For every parse tree, there exists a leftmost derivation
2. For every leftmost derivation, there exists a parse tree

Idea to prove 1. : (induction on the height of the tree)
Base case :

```
     A
   / | \
  a1 .. an
```

There is a rule $A \to a1 \ldots a_n$.

```
      A
    / | \
   /  |  \
  X1 ... Xn
  ~       ~
```

(with `~` a subtree) : $A \to X_1 \dots X_n$, and X_i and be derived in a leftmost fashion $X_i \xrightarrow{*} \omega_i$. By the induction hypothesis, we conclude.

Idea to prove 2. : induction on the number of steps of the derivation

Chomsky normal form (CNF)

A few rules are allowed:

$S \to \varepsilon$
$A \to BC$ (A, B, C are variables)
$A \to a$ where a is a terminal

$\omega = a_1 \dots a_N$ in $2N - 1$ steps : $N - 1$ steps to extend to N variables, then N steps to simplify the variables into terminals.

Theorem: context-free language (CNL) can be generated by a a grammar G in CNF.

Idea: convert a CFG G into normal form.

1. Add new S_0 : $S_0 \to S$
2. Remove ε-rules : assume $A \to \varepsilon$, $R \to uAv$. We add : $R \to uv$ (case $A \to \varepsilon$), we need to account all occurrences of A if $R \to uAvAw$. If $R \to A$, we add $R \to \varepsilon$ if $R \to \varepsilon$ have not been remove before.
3. Remove unit rules $A \to B$. For each $B \to u$, we add $A \to u$, if we had not deleted such a rule before, as previously.
4. We end up with $A \to u_1 \dots u_n$ with $u_i \in \Sigma \cup V$ or $S_0 \to \varepsilon$.
   $A \to u_1 \dots u_n$ goes to $A \to u_1 A_1$, $A_1 \to u_2 A_2 \, \dots A_{n-2} \to u_{n-1} u_n$. We eventually need to change rules if $u_i \in \Sigma$: $A_i \to U_i A_{i+1}$ and $U_i \to u_i$.

Pushdown automata

Definition

Definition : a PDA is a 6-tuple $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$, with :

- Q the set of states
- Σ the input alphabet
- Γ the tape alphabet
- $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$ the transition function
- q_0 the initial state
- F the set of accepting states

A PDA accepts $w = w_1 \dots w_M$ with $w_i \in \Sigma_\varepsilon$ if there are $r_0, \dots r_m \in Q$, $s_0, \dots, s_M \in \Gamma^*$ such that :

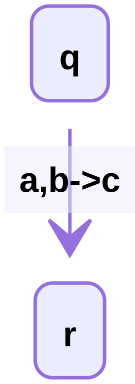
1. $r_0 = q_0$ and $s_0 = \varepsilon$
2. For $i = 0$ to $M - 1$:
   $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$

$$s_i = at, s_{i+1} = bt, \qquad a, b \in \Gamma_\varepsilon, t \in \Gamma^*$$

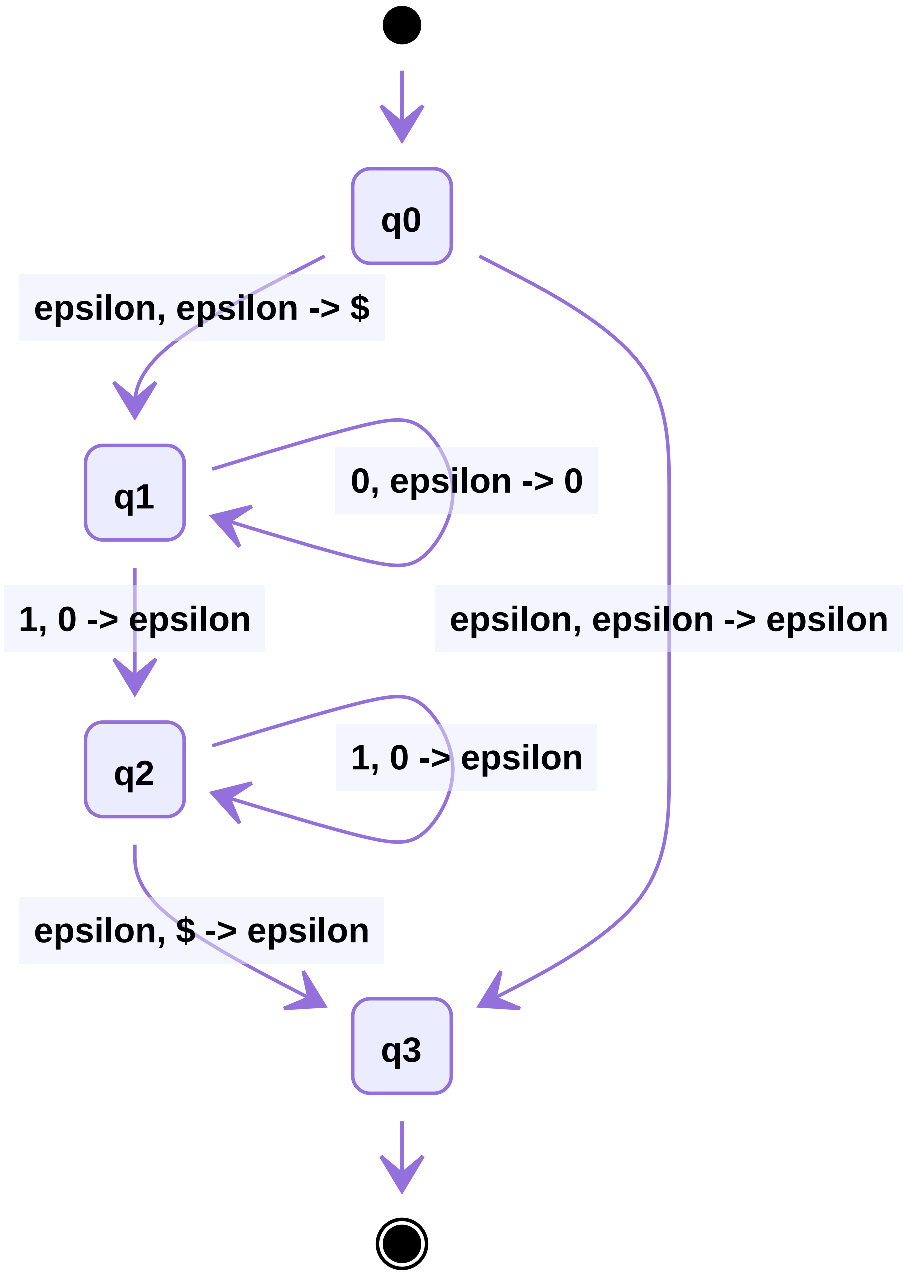
3. $r_M \in F(, s_M = \varepsilon)$

Remark: this is not a deterministic PDA. We could define a deterministic PDA, but it will only recognise unambiguous languages.

We will draw :



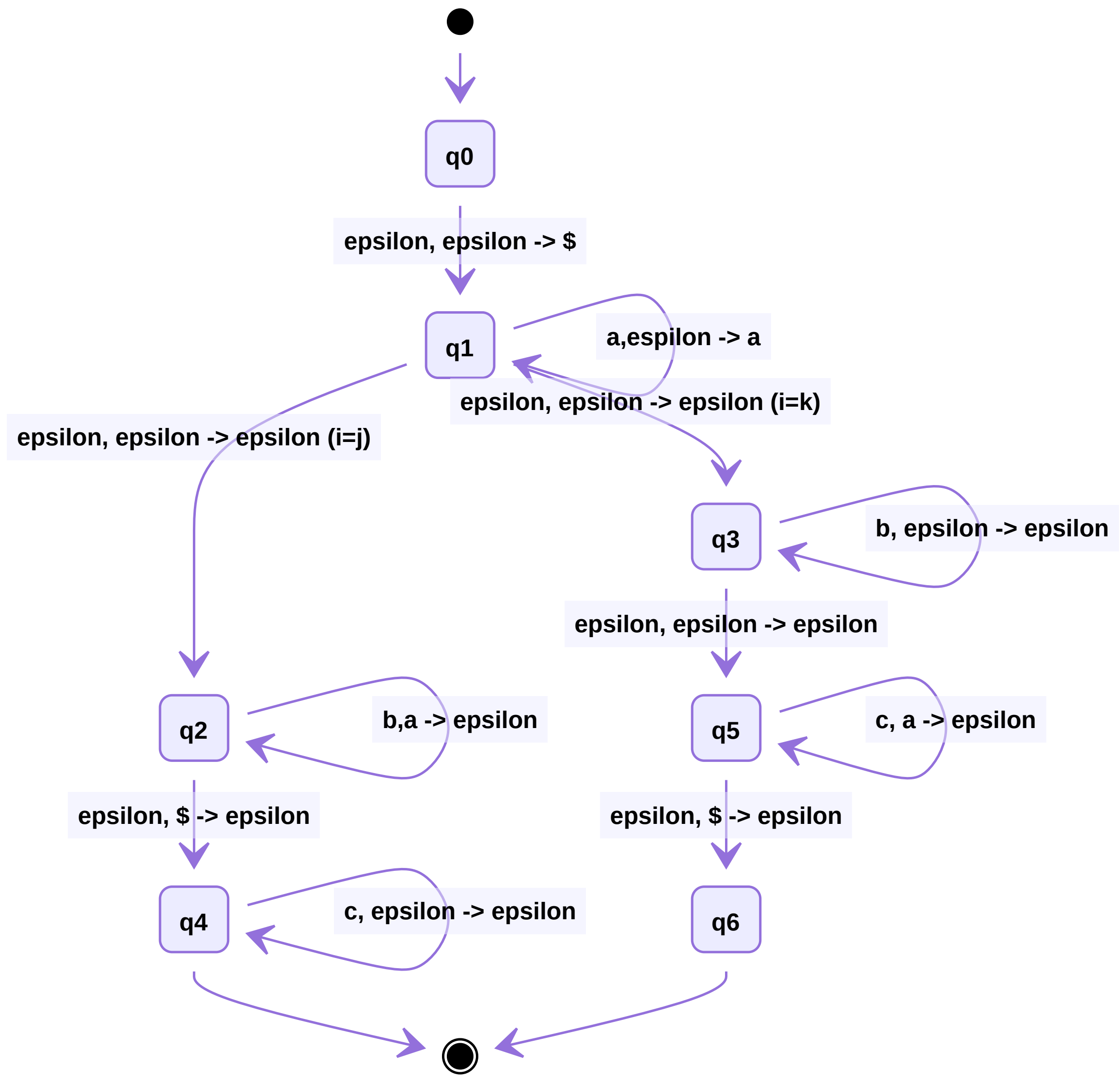
Meaning that we read a, and we replace b with c at the top of the stack ($\delta(q, a, b) \ni (r, c)$).



	0	1	ε
q_0			
q_1			
q_2			
q_3			

We can also put this information in a table like above, but this is harder to visualize.

Example : $L = \{a^i b^j c^k, i = j \text{ or } i = k\}$



Equivalence between PDA and CFG

From CFGs to PDAs

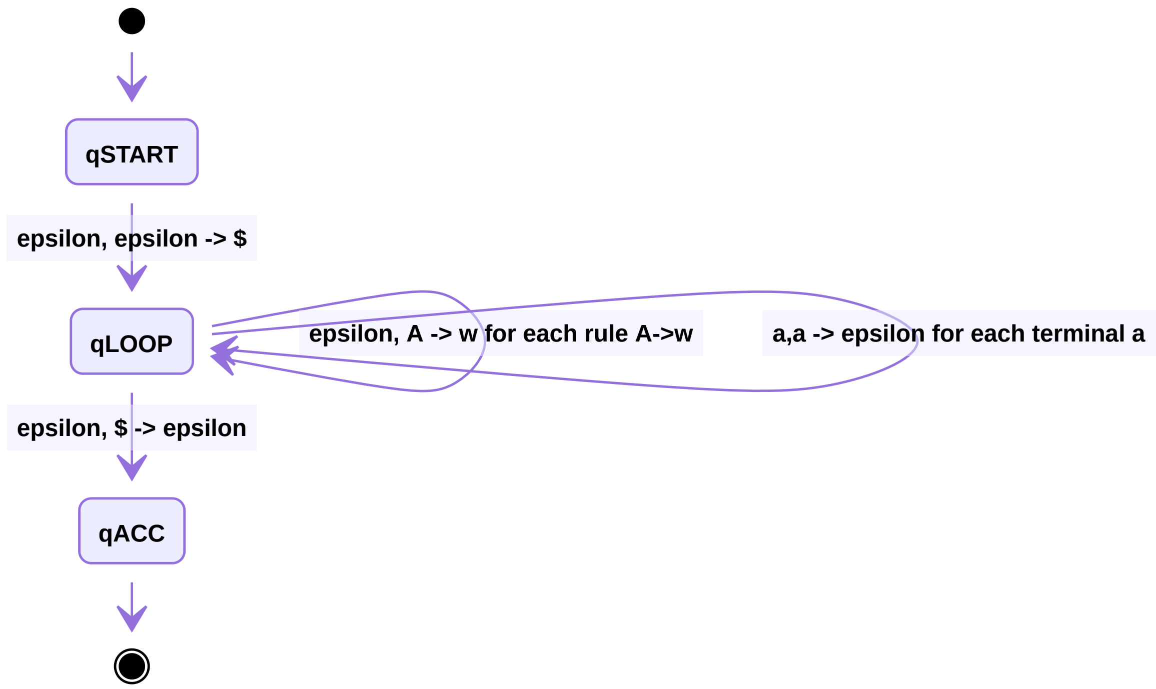
Claim: if L is a CFG, then some PDA recognises it.

Proof: $\exists$ a CFG G such that $L(G) = L$. We want to convert G into a PDA P.

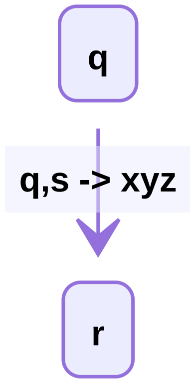
$S \rightarrow aTb|b$
$T \rightarrow Ta|\varepsilon$ => it converts into the stack as `[S;$] -> [a;T;b;$]`

If we read a a, it would become `[T;b;$]` then `[b;$]` for instance ($T \rightarrow \varepsilon$). We then read a b : `[$]`. `$` is on top, so it is a word accepted.

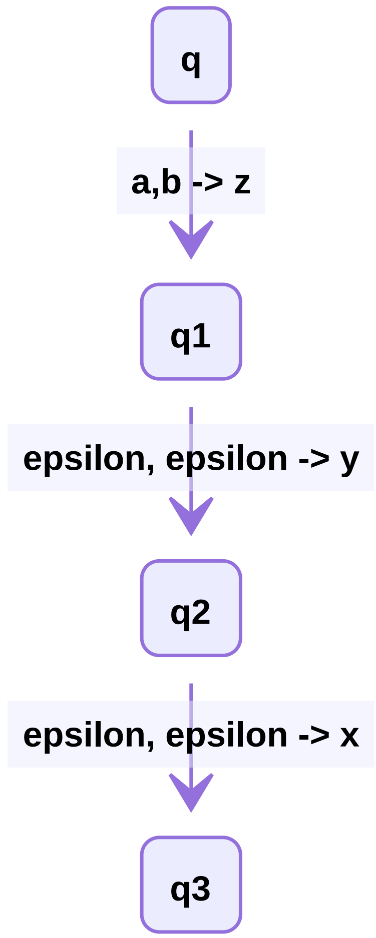
Thus, we can make the PDA like so :



We can then decompose each complex rule by using several states. For instance,



Can be decomposed into :

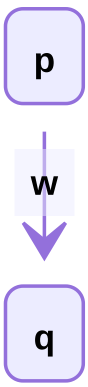


From PDAs to CGAs

Claim: if a PDA recognises a language L, then there exists a CFG G such that $L(G) = L$.

Proof: given a PDA P, we build a CFG G for $L(P)$.

We want the stack to be empty at the beginning and at the end (or more generally, if the stack is unchanged at the end compared to the beginning).



Goal : create a variable A_{pq} that can take P from p to q on an empty stack. We have $A_{pp} \to \varepsilon$.

Two situations :

1. the stack never gets empty between p and q.
2. the stack gets empty between p and q.

Assumption:

1. there is a single accept state
2. it empties before accepting
3. each transition is either a push or a pop

$$
\begin{array}{lll}
A_{pq} \to A_{pr} A_{rq} & & \forall p, q, r \in Q \\
A_{pq} \to a A_{rs} b & \text{if} & \begin{array}{r} \delta(p, a, \varepsilon) \ni (r, u) \\ \delta(s, b, u) \ni (q, \varepsilon) \end{array} \\
A_{p,p} \to \varepsilon & & \forall p \in Q
\end{array}
$$

Claim: if A_{pq} generates x then x can bring P from p to q with empty stacks.

Proof: base case : the derivation has 1 step. Then this rule is $A_{pp} \to \varepsilon$.

Induction step : assume this is true for k steps of derivation. Let's show that it is true for $k+1$ steps.

Suppose that A_{pq} generates x in $k+1$ steps.

Case 1: The first step is $\to A_{pq} \to aA_{rs}b$. $x = ayb$, $A_{rs} \overset{*}{\to} y$.

Then A_{rs} can bring P from r to s in k steps, by induction hypothesis.

Then the stack at r and s is `[u]`, so x can bring P from p to q on empty stacks.

Other case : see lecture notes.

Pumping lemma

Lemma: If L is a CFL, then there exists p such that for every $s \in L$, $|s| \geq p$, that can be broken into $s = uvwxyz$ such that

1. for each $i : uv^i xy^i z \in L$
2. $|vy| > 0$
3. $|vxy| \leq p$

Proof: let s be "very long" (in a sense to be defined later on). Then the parse tree must be "very tall".

cf lecture notes

Example: $L = \{a^N, b^N, c^N, N \geq 0\}$.

We assume L is a CFL. Then there exists p such that $a^p b^p c^p \in L$. There exists u, v, x, y, z such that $a^p b^p c^p = uvwxyz$ and $uv^i xy^i z \in L \quad \forall i \geq 0$. We then, by case enumeration, conclude that this is not possible.

Turing machines

Turing machines

- Used an infinite tape as its unlimited memory
- Has a tape head
  - Can read and write symbols
  - Can move left or right
- Initially, the tape contains only the input string and is blank everywhere else
- Has an accept and a reject state

- The outputs `accept` and `reject` are obtained by entering these states
- It will go on forever if it does not enter an accepting or a rejecting state

Differences with a finite automate:

- A Turing machine can both write on the tape and read from it
- The read-write head can move both to the left and to the right
- The tape is infinite
- Rejecting and accepting states take effect immediately

Turing machine M_1 for the language $B = \{w\#w | w \in \ 0, 1^*\}$

M_1 : on input string w:

1. Zig-zag across the tape to match symbols
   - if symbols do not match or if no $\#$ is found, reject
   - Cross of symbols as they are checked
2. When all symbols to the left of the $\#$ have been crosses off
   - Check for any remaining symbols to the right of the $\#$
   - If any symbols remain, reject. Otherwise, accept.

Formal definition:

A Turing machine is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ where Q, Σ, Γ are finite sets and

1. Q is the set of states
2. Σ is the input alphabet not containing the blank symbol $\sqcup$.
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$.
4. $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is the transition function
5. $q_0 \in Q$ is the start state.
6. $q_{accept} \in Q$ is the accept state.
7. $q_{reject} \in Q$ is the reject state, where $q_{reject} \neq q_{accept}$.

Turing machine computation

- Initially :
  - Input $w = w_1 \dots w_n \in \Sigma^*$ on the leftmost n squares of the tape
  - The rest of the tape is blank
  - The head starts on the leftmost square
  - First blank appearing in the tape marks the end of the input.
- Computation proceeds according to the rules
  - Head stays in the same place if trying to move to the left off the left-hand end
- ...
- When TM computes, the current state, tape contents, and head location change
- A TM configuration describes the setting of these three items
- A configuration C is represented as uqv
  1. Current state is q
  2. Tape contents is uv where $u, v \in \Gamma^*$
  3. The current head location is the first symbol of v

Derivation of a configuration

- A configuration C_1 yields a configuration C_2 (denoted $C_1 \vdash_M C_2$) if M can legally go from C_1 to C_2 in a single step

  - If $C_1 = uaq_ibv$ and $C_2 = uq_jacv$ and $\delta(q_i, b) = (q_j, c, L)$
  - If $C_1 = uaq_ibv$ and $C_2 = uacq_jv$ and $\delta(q, j, b) = (q_j, c, R)$
  - If $C_1 = q_ibv$ and $C_2 = q_jcv$ and $\delta(q_i, b) = (q_j, c, L)$
  - If $C_1 = bvq_i$ and $C_2 = uacq_j$ and $\delta(q_i, b) = (q_j, c, R)$
- We write $C \vdash_M^* C'$ the transitive closure of $\vdash$.

Configuration types:

- Start configuration on M on input w : q_0w
- uqv is an accepting configuration if $q = q_{accept}$
- Idem for rejecting

Language of a Turing Machine

- The language recognised by M, denoted $L(M)$ is the collection of strings that M accepts.
- A language is called Turing-recognisable (a.k.a recursively enumerable) if some Turing machine recognises it
- A Turing machine is called a decider if it always halts
- A language is Turing-decidable if some Turing machine decides it.

Variants of Turing machines

- Equivalent:
  - Multiple tapes
  - Taps that are bi-infinite
  - Non-determinisms
- Other models are TM-equivalent if they satisfy reasonable requirements
  - Unrestricted and unlimited memory
  - Finite amount of work in each step (computation is local)
  - Finite set of instructions
- Church-Turing Hypothesis : λ-calculus ($\approx$ intuitive computation notion) $\equiv$ Turing machines

Multi-tape TM

- A multi-tape TM is similar to an ordinary one with many tapes
  - Each tape has its own head for reading and writing
  - The input appears initially on tape 1
  - Other tapes start initially blank
  - Transition function allows for reading, writing, and moving the heads on all tapes simultaneously
  - $\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R\}^k$
- We can simulate a multi-tape TM M with a single tame TM S : we concatenate the k tapes onto one, separating each one with a $\#$.
  - Dotted symbols to mark the head positions

Non-deterministic Turing machines (NTM)

- The machine may proceed according to several possibilities at any point
- Transition function has the form $\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$
- Allows branching of the computation
- The machine accepts its input if some branch accepts

Equivalence between TMs and NTMs

- We can simulate N a NTM with a 3-tape TM D

- Try all possible branches of N's non-deterministic computation
- If D finds the accept state on one branch, D accepts
- Otherwise, D's simulation will not terminate
- Use BFS to avoid getting caught in an infinite branch.
    1. Input tape
    2. Simulation tape
    3. Address tape

Decidable languages

- A NTM is a decider if all branches halt on all inputs.
- The simulator D will always halt if N is a decider
- Example of decidable languages:
    - $L_{conn} = \{< G > | G \text{ is a connected undirected graph}\}$ where $< G >$ means an encoding of G
    - $L_{CFG} = \{< G > | G \text{ is a CFG and } L(G) = \varnothing\}$

Enumerators

- An enumerator E is a Turing machine with an attached printer
    - E can add a string to the list by sending it to the printer
    - E starts with a blank input on its work tape
    - If E does not halt, E may print an infinite list of string
    - E's language is the set of strings that it eventually prints out
    - E may generate the strings in any order, possibly with repetitions.

Recursively enumerable languages

- If L is Turing-recognisable, then we can enumerate the strings in L
    - Let M be a TM that recognises L
    - Let $s_1, \ldots$ be a list of all possible strings in Σ^*
    - We can build an enumerator E as follows
    - E = "ignore the input
        1. "Repeat the following for $i = 1, \ldots$
        2. Run M for i steps on each input $s_1, \ldots, s_i$
        3. If any computations accept, print out the corresponding s_i"
    - This is why these languages are also known as recursively enumerable
- If E enumerates a language A, then there is a TM M that recognises A
- M = "On input w:
    1. Run E. Every time that E outputs a string, compare it with w
    2. If w ever appears in the output of E, accept"

Universal TMs

- A universal TM U is a TM that, on input $< M, w >$, simulates M on w and returns the output
- U will loop if M loops on input w
- U can simulate M as follows

- Assume M is single-tape and $\Sigma = \{0, 1, \sqcup\}$
- U writes $q_0 w$ on its work tape, where $w = w_1 \dots w_n$
- U scans the description if M until it finds $\delta(q_0, w_1)$
- M applies it to tape contents
- M continues to next transition

Closure proprieties

- If L_1 and L_2 are Turing-recognisable (resp. decidable), so are
  - $L_1 \cup L_2$
  - $L_1 \cap L_2$
  - $L_1 L_2$
  - L_1^*
  - L^R (reverse language)
- If L_1 and L_2 are Turing-recognisable :
  - $L_1 \setminus L_2$ may not be
  - $L_1^\complement$ may not be

Undeciadability

Detour: let $S = \{s_1, \dots, s_n\}$ and $P = \{p_1, \dots, p_m\}$ be two finite sets. There exists a bijection between S and P iff $|S| = |P|$. What if S and P are infinite ?

We say that two sets S and P have the same size if there is a bijection $f : S \to P$. Ex : bijection between $S = \mathbb{N}$ and $P = 2\mathbb{N}$.

A set is countable iff it is of size less of equal to the size of $\mathbb{N}$. Ex : $\mathbb{Q}$ is countable (because $\mathbb{N} \times \mathbb{N}$ is and $\mathbb{Q}$ and $\mathbb{N} \times \mathbb{N}$ are trivially in bijection).

On the other hand, $\mathbb{R}$ is uncountable (argument : Cantor's diagonal).

Some languages are not Turing-recognizable.

- The set of all Turing machines is countable
  - Set of all strings Σ^* is countable for any alphabet Σ.
  - Each Turing machine M has an encoding into a string $< M >$.
- The set of all languages is uncountable
  - Let $\mathcal{B}$ of all binary is uncountable
  - Let $\mathcal{L}$ be the set of all languages over alphabet Σ.
  - Let $\Sigma^* = \{s_1, s_2, \dots\}$.
  - Each language $A \in \mathcal{L}$ has a unique sequence in $\mathcal{B}$. (presence or absence of s_i in the language).
  - Hence, $\mathcal{L}$ is uncountable.

A_{TM} is undeciable.

- Assume $A_{TM} = \{< M, w > | M \text{ is a TM which accepts } w\}$ is decidable.
- Suppose that H is a decider for A_{TM}.
- Construct a TM D with H as a subroutine.
  - $D =$"on input $< M >$, where M is a TM :
    1. Run H on input $< M, < M >>$
    2. If H accepts, reject

3. If H rejects, accept"

- By construction, no matter the input, we get a contradiction, a la Russel.

Claim: $\mathcal{L}$ is decidable iff $\mathcal{L}$ is Turing-recognisable and co-Turing-recognisable ($\bar{\mathcal{L}}$ is recognised).

$\Rightarrow$: trivial

$\Leftarrow$: Let M be the recogniser for $\mathcal{L}$ and $\bar{M}$ be the recogniser for $\bar{\mathcal{L}}$.

$\hat{M}$ = on input w:

- Run both M and $\bar{M}$ on w in parallel.
- If M accepts w, accept, if $\bar{M}$ accepts w, reject.

Corollary: $\overline{A_{TM}}$ is not Turing-recognisable.

Further examples of undeciddable languages

- $HALT_{TM} = \{< M, w >, M$ is a TM which halts on $w\}$.
- E_{TM} = set of $< M >$ such that M is a TM and $\mathcal{L}(M)$ not empty...

The halting problem

- Proof by contradiction.
- Let $HALT_{TM} = \mathcal{L}(R)$ for some decider R.
- Construct a decider S for A_{TM} using R as a subroutine.

S = "on input $< M, w >$, where M is a TM,

1. Run R on input $< M, w >$
2. If R rejects, reject
3. If R accepts, simulate M ow w until it halts
4. If M has accepted, accept, if M has rejected, reject."

Since A_{TM} is undecidable, R cannot exit.

Other undecidable problems

Undecidability of $E_{TM} = \{< M >, \mathcal{L}(M) \neq \varnothing\}$.

- Proof by construction
- Let $E_{TM} = \mathcal{L}(R)$ for some decider R.
- Define M_E=" on input x:
    - If $x \neq w$: reject
    - If $x = w$, run $M(w)$ and accept if M does"
- Therefore
    - $\mathcal{L}(M_E) = \{w\}$ if $M(w)$ accepts
    - $\mathcal{L}(M_E) = \varnothing$ if $M(w)$ rejects.
- Constructing a decider S for A_{TM} using R.
- S = "on input $< M, w >$:
    1. "Construct M_E as described.
    2. Run R on input $< M_E >$
    3. If R accepts, reject
    4. If R rejects, accept.
- Since A_{TM} is undecidable, R cannot exit.

Undecidability of $REGULAR_{TM} = \{< M >$, M is a TM and $\mathcal{L}(M)$ is a regular language$\}$.

- Proof by contradiction
- Let $REGULAR_{TM} = \mathcal{L}(R)$ for some decider R.
- Define M_r = "on input x:
  - If $x = 0^n1^n$, accept.
  - If $x \neq 0^n1^n$; run $M(w)$ and accept if M does".
- Therefore
  - $\mathcal{L}(M_r) = \{0^n1^n, n \geq 0\}$ if $M(w)$ rejects
  - $\mathcal{L}(M_r) = \sigma^*$ if $M(w)$ accepts.
- Constructs a decider S for A_{TM} using R.
- $S =$" on input $< M, w >$
  1. Construct M_r as described.
  2. Run R on input $< M_r >$.
  3. If R accepts, accept.
  4. If R rejects, reject.
- Since A_{TM} is undecidable, R cannot exit.

Undecidability of $EQ_{TM} = \{< M_1, M_2 >, \mathcal{L}(M_1) = \mathcal{L}(M_2)$ and M_1 and M_2 TMs $\}$

- Proof by contradiction.
- Let $EQ_{TM} = \mathcal{L}(R)$ for some decider R.
- Define $M_{rej} =$ on input x: reject
- Construct a decider S for A_{TM} using R:
- S = on input $< M >$ where M is a TM:
  1. Construct M_{rej} as above.
  2. Run R on input $< M, M_{rej} >$.
  3. If R accepts, accept.
  4. If R reject, reject.
- Since E_{TM} is undecidable, R cannot exit.

Mapping reducibility

Definition: a function $f : \Sigma^* \rightarrow \sigma^*$ is computable if some Turing machine M, on every input w, halts with just $f(w)$ on its tape.

Definition: Language A is mapping reducible to language B, written $A \leq_m B$, if there is a computable function $f : \Sigma^* \rightarrow \Sigma^*$ where $\forall w, w \in A \Leftrightarrow f(w) \in B$.

f is then called the reduction from A to B.

Relations

Theorem: if $A \leq_m B$ and B is decidable, then A is decidable.

Proof:

- Let M be the decider for B and f be the reduction from A to B.
- We can build a decider N for A as follows:

- N = "on input w:
    1. Compute $f(w)$
    2. Run M on $f(w)$ and output whatever M outputs."

Corollary: If $A \leq_m B$ and A is undecidable, then B is undecidable.

Example: $A_{TM} \leq_m HALT_{TM}$

- Must provide a computable function f that takes input of the form $< M, w >$ and returns output of the form $< M', w' >$, where $< M, w > \in A_{TM} \Leftrightarrow < M', w' > \in HALT_{TM}$
- The following machine F computes a reduction f.
- $F =$ "on input $< M, w >$: "<
    1. Construct the following machine M':
    2. M="on input x":
        1. Run M on x
        2. If M accepts, accept
        3. If M rejects, enter a loop
    3. Output $< M', w' >$.

Example: EQ_{TM} is not Turing-recognisable.

Claim: $A_{TM} \leq_m EQ_{TM}$.

- Must provide f such that $< M, w > \in A_{TM} \Leftrightarrow < M_1, M_2 > \in EQ_{TM}$
- The following machine F computes a reduction f.
- $F =$ "on input $< M, w >$ where M is a TM and w is a string"
-
    1. Construct M_1
        1. M_1 = accept on all inputs.
    2. Construct M_2
        1. M_2 = on any input:
            1. Run M on w
            2. If it accepts, accept.
    3. Output $< M_1, M_2 >$".

Example : $\overline{EQ_{TM}}$ is not Turing-recognisable.

Claim: $A_{TM} \leq_m \overline{EQ_{TM}}$. Same idea as above (invert accept and reject on M_1.)