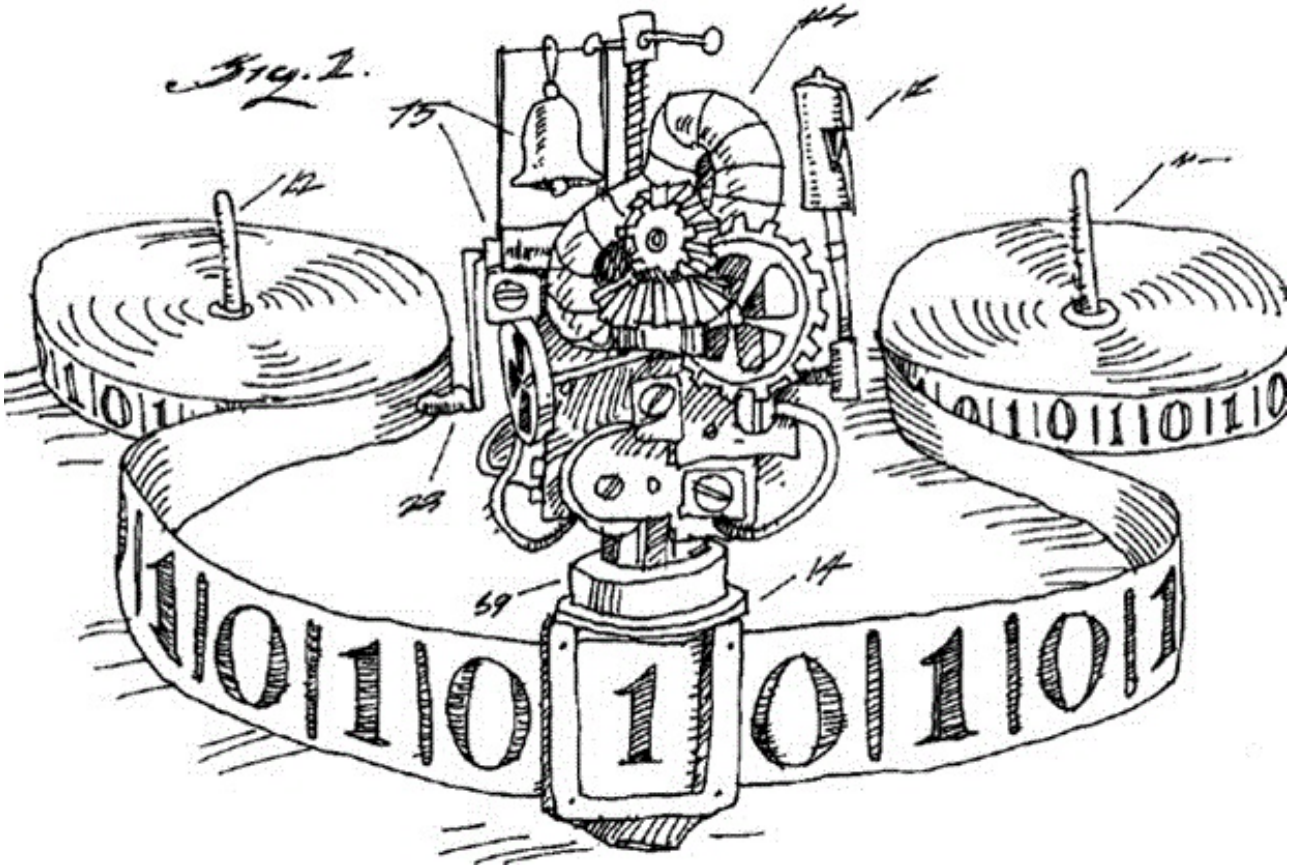
Chapitre 3

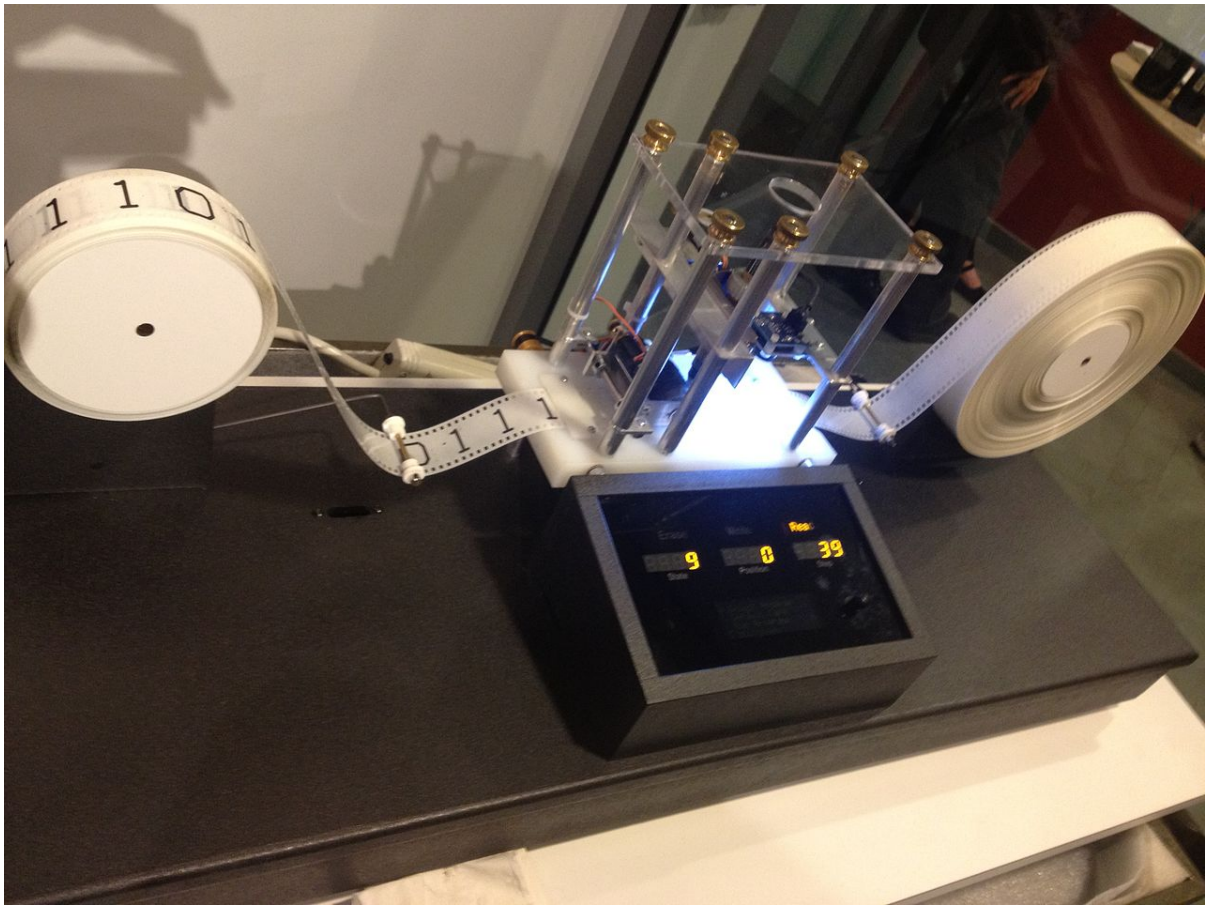
Thèse de Church-Turing

Alan Turing

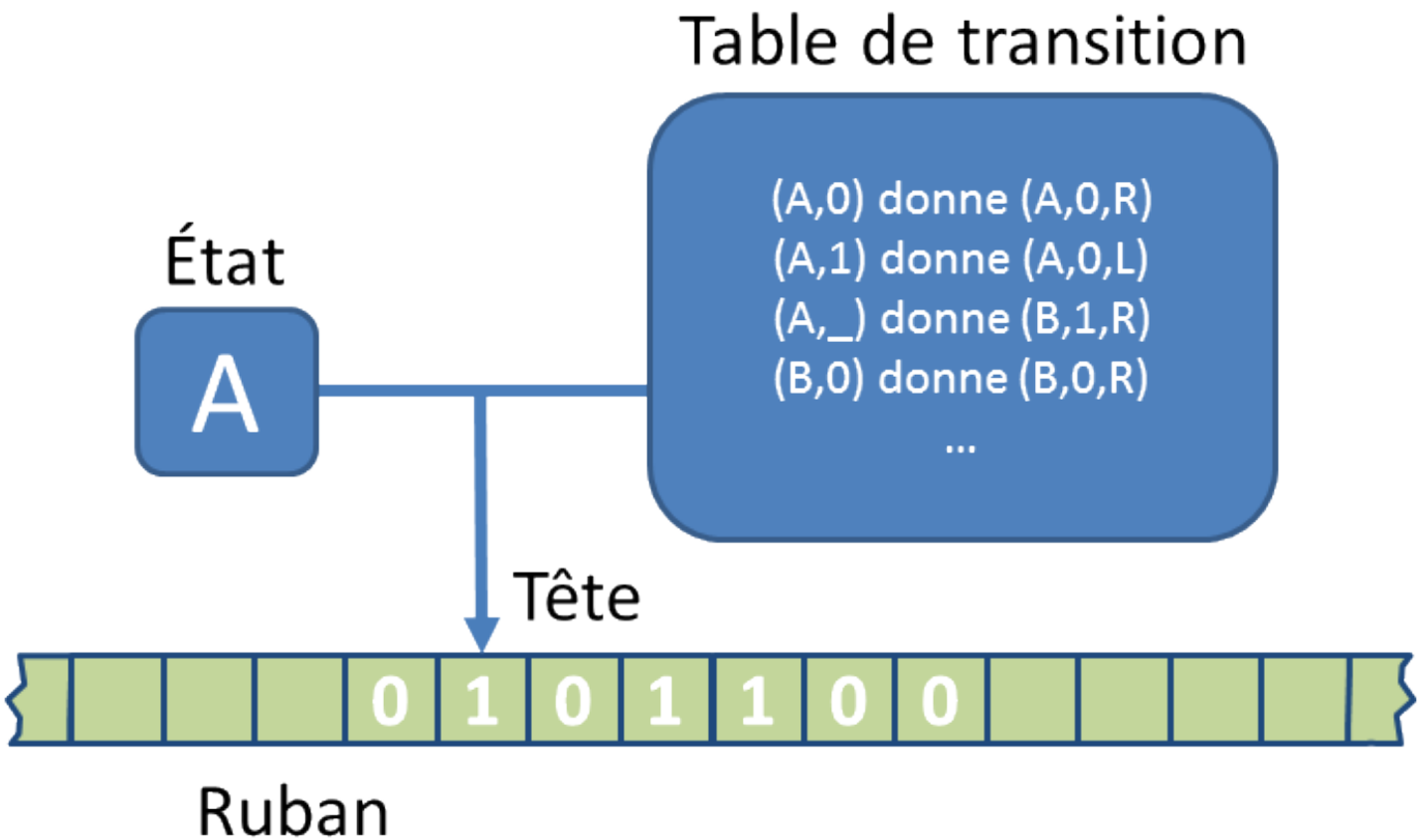
Les machines de Turing



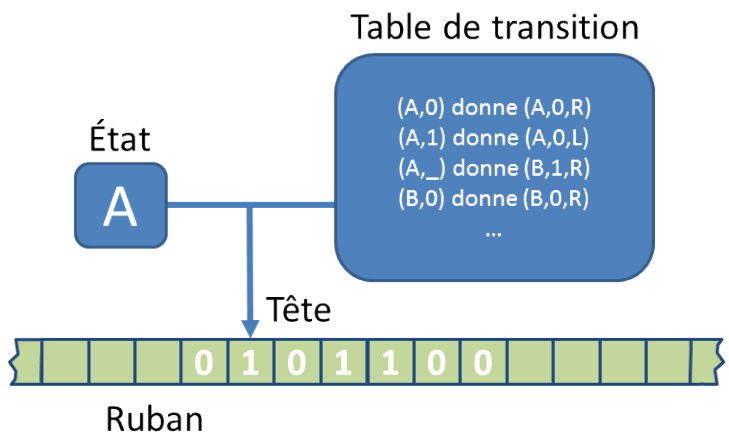
Les machines de Turing



Les machines de Turing



- Une machine de Turing peut lire le contenu du *ruban* et écrire sur celui-ci.
- Le *contrôle* passe d'un *état* à un autre à chaque étape du calcul et il y a un nombre fini d'états possibles.
- À chaque étape de calcul, la *tête* de lecture et d'écriture peut bouger à gauche ou bouger à droite.
- Le ruban est illimité vers la droite et vers la gauche.
- Si le contrôle passe à un état *final*, alors le calcul est terminé.



Définition formelle d'Alan Turing en 1936 :

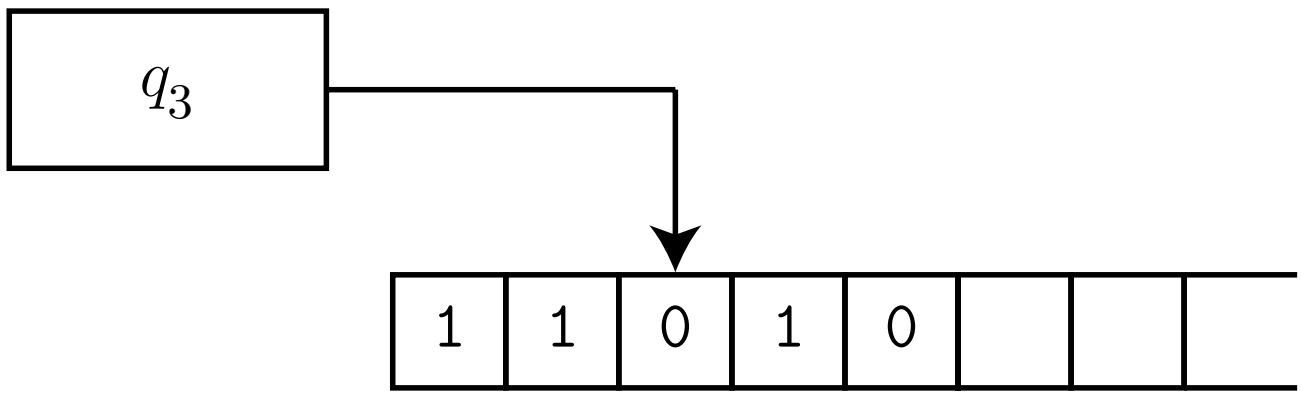
Définition 3.1

Une machine de Turing (MT) est un 6-tuplet $(Q, \Sigma, \Gamma, \delta, q_0, F)$ où :

- Q est un ensemble fini d'états ;
- Σ est l'alphabet ;
- Γ est l'alphabet de ruban, $\sqcup \in \Gamma$ et $\Sigma \subseteq \Gamma$;
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\langle \textsc{Gauche} \rangle, \langle \textsc{Droite} \rangle\}$ est la fonction de transition ;
- q_0 est l'état initial ;
- $F = \{q_A, q_R\}$ sont les états finaux.

Définition 3.2

Une configuration de la MT $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ est un triplet (u, q, v) avec $q \in Q$ et $u, v \in \Gamma^*$, ce qui s'interprète en disant que le ruban de M contient uv, suivi et précédé d'une chaîne infinie de $\sqcup$. De plus, la tête est au-dessus du premier symbole de v.



La configuration $11q_3010$.

Remarque 3.3

Même si le ruban d'une MT est mathématiquement infini, nous dirons que le contenu du ruban est la chaîne finie $v \in \Gamma^*$ si le ruban contient v, suivi et précédé d'une chaîne infinie de $\sqcup$, pour v ne commencant ni ne terminant par $\sqcup$.

De la même façon, nous parlerons de l'extrémité droite du ruban, ou du dernier caractère du ruban, etc.

Définition 3.4

Pour $q, q' \in Q$, $u, v \in \Gamma^*$ et $x, x', z \in \Gamma$, on dit que la MT $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ passe de la configuration $c = (uz, q, xv)$ à la configuration $c' = (u, q', zx'v)$ si $\delta(q, x) = (q', x', \langle \textsc{Gauche} \rangle)$.

La machine M passe de la configuration $c = (u, q, xv)$ à la configuration $c' = (ux', q', v)$ si $\delta(q, x) = (q', x', \langle \textsc{Droite} \rangle)$.

On dit qu'il s'agit d'une transition de c à c'.

Une telle transition est notée $c \vdash c'$.

On écrit $c \overset{*}{\vdash} c'$ si M passe de c à c' en 0, 1 ou plusieurs transitions successives.

Définition 3.5

Soient un mot $w \in \Sigma^*$ et une machine de Turing $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$.

On dit que M accepte w si

$$(\varepsilon, q_0, w) \overset{*}{\vdash} (u, q_A, v).$$

On dit que M rejette w si

$$(\varepsilon, q_0, w) \overset{*}{\vdash} (u, q_R, v).$$

Sinon, on dit que M boucle sur w.
(M ne s'arrête pas)

Définition 3.6

Soient L un langage sur alphabet Σ et $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ une MT.

On dit que M décide L si $\forall w \in L, M$ accepte w et si $\forall w \notin L, M$ rejette w.

Définition 3.7

Pour $w \in \Sigma^*$, $w' \in \Gamma^*$ et $u \in \Gamma^*$, on dit que la MT $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ retourne ou s'arrête sur le mot w' sur entrée w si

$$(\varepsilon, q_0, w) \overset{*}{\vdash} (u, q, w')$$

pour $q \in F$. Sinon, si M n'atteint jamais un état final à partir de la configuration (ε, q_0, w), alors on dit que M boucle sur entrée w.

Définition 3.8

Soient une fonction

$$f \ : \ \Sigma^* \to \Sigma^*$$

et une MT

$$M = (Q, \Sigma, \Gamma, \delta, q_0, F).$$

On dit que f est calculée par M si pour tout $w \in \Sigma^*$, on a :

$$(\varepsilon, q_0, w) \overset{*}{\vdash} (u, q_F, f(w)) \quad \text{pour un certain } u \in \Gamma^*.$$

On dit aussi que M calcule ou implante la fonction f.

Descriptions des MT

Les nombreux exemples de MT faits précédemment donnent un aperçu des tâches qu'elles peuvent accomplir.

D'autre part, la plupart de ces tâches, même les plus simples, sont exécutées par des MT dont les définitions formelles sont abstruses et difficilement compréhensibles.

À partir de maintenant, nous décrirons les MT en langage naturel. Les descriptions seront schématiques, mais suffisamment claires et détaillées pour qu'on puisse sans difficulté donner la définition formelle correspondante, ou pour se convaincre qu'une telle définition existe bel et bien.

L'exemple suivant décrit une MT qui ajoute 1 à un nombre naturel écrit en binaire.

Exemple 3.9

PLUS-UN :

1. *Prendre un nombre naturel en binaire en entrée ;*
2. *Parcourir le ruban de gauche à droite*
3. *Si ce chiffre est 0, alors le remplacer par 1 ;*
4. *Sinon, remplacer tous les 1 consécutifs par des 0, de droite à gauche, jusqu'au premier 0 rencontré, et remplacer ce 0 par 1 ; si on atteint une case vide ⊔ à la fin du ruban avant de rencontrer un 0, alors écrire 1 sur celle-ci.*

Les MT peuvent simuler les programmes TANTQUE.

Théorème 3.10

Soit

$$f : \mathbb{N} \to \mathbb{N}$$

une fonction calculée par un programme TANTQUE. Alors il existe une MT M qui calcule une fonction

$$f' : \{0,1\}^* \to \{0,1\}^*$$

telle que $f(x) = y$ implique $f'(x') = y'$, où x' et y' sont les représentations binaires de x et y respectivement.

Aperçu de la démonstration :

Pour que M puisse simuler un programme `TANTQUE`, elle doit pouvoir travailler avec des registres contenant des nombres entiers, c'est-à-dire entre autres :

- incrémenter la valeur d'un registre ;
- déterminer si deux registres sont égaux ou non.

La première de ces tâches est illustrée à l'exemple 3.9, et la deuxième est laissée en exercice.

Les registres seront présents sur le ruban de M sous forme binaire et séparés par le caractère $\#$.

La machine M commence par insérer les caractères 0# au début du ruban, ce qui crée le registre r_0 et l'initialise à 0.

Les registres de travail $r_1, \ldots, r_l$ sont créés et initialisés en ajoutant #0 pour chacun d'eux à la fin du ruban.

Les branchements nécessaires à l'exécution du programme `TANTQUE` sont implantés à l'aide des états de M, c'est-à-dire que différents points dans le programme correspondent à différents états de M.

Lorsque la simulation est terminée, M efface les registres de travail, ramène la tête de lecture en première position, et passe à son état final. □

Remarque 3.11

Le théorème 3.10 peut être facilement généralisé aux cas où la fonction f peut prendre la valeur $\uparrow$, c'est-à-dire les cas où le programme peut boucler à l'infini, de même pour les cas où f est une fonction de plusieurs variables entières.

Variantes des machines de Turing

Plusieurs variantes des machines de Turing peuvent être simulées de façon similaire :

- Machines à plusieurs rubans
- Machines à plusieurs *pistes* par ruban

De plus, on peut se restreindre à un alphabet binaire, en divisant le ruban en blocs.

Les programmes TANTQUE peuvent simuler les MT.

Théorème 3.12

Soit

$$f \ : \ \Sigma^* \to \Sigma^*$$

une fonction calculée par une MT $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$. *Alors il existe un programme* TANTQUE *qui calcule une fonction*

$$f' \ : \ \mathbb{N} \to \mathbb{N}$$

telle que $f(x) = y$ *implique* $f'(x') = y'$, *où les entiers* x' *et* y' *sont les représentations (dans un codage de Gödel) des chaînes* x *et* y *respectivement.*

Aperçu de la démonstration :

On peut voir le ruban de M comme un tableau infini de caractères dans Γ de la forme

$$(c_1, c_2, \dots, c_n, \sqcup, \sqcup, \dots).$$

Si on réserve l'entier 0 pour le caractère $\sqcup$, le ruban peut être représenté par un tableau infini d'entiers de la forme

$$(a_1, a_2, \dots, a_n, 0, 0, \dots),$$

ou, grâce au codage de Gödel, par l'entier

$$p_1^{a_1} p_2^{a_2} \dots p_n^{a_n}.$$

Le registre r_1 va donc contenir le ruban de M sous cette forme.

Le registre r_2 contient la position de la tête de lecture.

Le registre r_3 contient l'état de M, encodé dans un entier.

Chaque transition de M peut être facilement simulée dans les registres r_1, r_2 et r_3, à l'aide, entre autres, des programmes `TABLVAL` et `TABLASS` vus au chapitre précédent.

Lorsque la simulation est terminée, la portion du ruban de M qui contient $f(x)$ est copiée dans le registre r_0, toujours en codage de Gödel. $\square$

Remarque 3.13

Le théorème 3.12 peut facilement être généralisé aux cas où M peut boucler à l'infini.

Remarque 3.14

Les théorèmes 3.10 et 3.12 nous permettent de décrire des MT comme des programmes `TANTQUE`, si cela est utile.

Alonzo Church

Le λ-calcul

Le λ-calcul est un modèle de calcul défini par Alonzo Church en 1936 (en même temps que Turing !). Il est basé sur les λ-termes :

Définition 3.15 (λ-terme)

L'ensemble des λ-termes, dénoté Λ, est défini récursivement comme suit :

- Variables : Les variables $x, y, z, \cdots$ sont des λ-termes.
- Applications : Si $M \in \Lambda$ et $N \in \Lambda$, alors $(M\ N) \in \Lambda$.
- λ-abstractions : Si x est une variable et $M \in \Lambda$, alors $(\lambda x.M) \in \Lambda$.

Interprétation : $\lambda x.M$ est la fonction anonyme $x \mapsto M$. Les fonctions peuvent prendre des fonctions comme arguments.

Les opérations suivantes sont permises sur les λ-termes :

- α-réduction : On peut renommer les arguments des fonctions : $\lambda x.M$ est équivalent à $\lambda y.M[x := y]$. Ici, $M[x := y]$ dénote l'expression M où on a remplacé toutes les instances de x par y.
- β-réduction : $((\lambda x.M)N)$ est équivalent à $M[x := N]$.

Ces propriétés permettent d'interpréter $\lambda x.M$ comme la fonction $x \mapsto M$.

Déroulement d'un calcul : on fait des β-réductions jusqu'à ce qu'on ne puisse plus. Théorème de Church-Rosser : l'ordre dans lequel on fait les réductions ne change rien au résultat.

On peut simplifier la notation :

- MNR veut dire $(MN)R$
- $\lambda x.MN$ veut dire $\lambda x.(MN)$
- $\lambda xy.M$ veut dire $\lambda x.\lambda y.M$

Cette troisième propriété sert à représenter les fonctions à deux arguments (ou plus) : une fonction à deux arguments $(x, y) \mapsto M$ peut être vue comme la fonction $x \mapsto (y \mapsto M)$.

Donc, on a des fonctions abstraites, mais pas de données ni d'opérations de base ! Il faut donc encoder des données seulement avec la définition de base.

Les nombres naturels, avec l'encodage de Church :

$$
\begin{aligned}
0 &:= \lambda f.\lambda x.x \\
1 &:= \lambda f.\lambda x.fx \\
2 &:= \lambda f.\lambda x.f(fx) \\
3 &:= \lambda f.\lambda x.f(f(fx)) \\
&\vdots
\end{aligned}
$$

0 est donc une fonction qui prend la fonction f en entrée, la jette aux poubelles, et retourne la fonction identité $\lambda x.x$.

1 prend f en entrée et retourne la fonction $\lambda x.(fx)$ qui applique f une fois (et qui est donc égale à f).

On peut voir que n est une fonction qui prend f en entrée, et retourne la fonction $f^{\langle n \rangle}$.

On peut définir des opérations sur les nombres naturels :

$$
\begin{aligned}
\texttt{INC} &:= \lambda n.\lambda f\ x.f\ (n\ f\ x) \\
\texttt{PLUS} &:= \lambda m\ n.\lambda f\ x.m\ f\ (n\ f\ x) \\
\texttt{MULT} &:= \lambda m\ n.\lambda f.m\ (n\ f) \\
&= \lambda m\ n.m(\texttt{PLUS}\ n)\ 0 \\
\texttt{EXP} &:= \lambda m\ n.n\ m
\end{aligned}
$$

La décrémentation et la soustraction sont nettement plus durs à définir ; nous y reviendront dans quelques slides.

On peut également définir les booléens :

$$\texttt{VRAI} := \lambda x\ y.x$$
$$\texttt{FAUX} := \lambda x\ y.y,$$

avec les opérations logiques :

$$\texttt{NEG} := \lambda p.p\ \texttt{FAUX}\ \texttt{VRAI}$$
$$\texttt{ET} := \lambda p\ q.p\ q\ p$$
$$\texttt{OU} := \lambda p\ q.p\ p\ q$$
$$\texttt{SAS} := \lambda p.p$$

On peut construire des structures de données complexes :

$$\texttt{nTUPLE} := \lambda r_1 \ldots r_n.\lambda f.f\ r_1\ \ldots\ r_n,$$

donc $(\texttt{2TUPLE}\ x\ y)\ f = (f\ x\ y)$. Pour extraire le $k$ième élément d'un `nTUPLE` :

$$\texttt{ELEMk} := \lambda t.t\ (\lambda r_1 \ldots r_n.r_k),$$

donc par exemple $\texttt{ELEM2}\ (\texttt{2TUPLE}\ x\ y) = y$.

On peut utiliser 2TUPLE pour construire une « liste chaînée » :

$$\ell = \texttt{2TUPLE}\ x\ \ell'$$

où x est le premier élément, et ℓ' est le reste de la liste. Une liste vide peut alors être définie comme

$$\texttt{VIDE} := \lambda z.\texttt{VRAI},$$

et on peut tester si une liste est vide via :

$$\texttt{VIDE?} := \lambda \ell.\ell\ (\lambda z w.\texttt{FAUX}).$$

Ceci nous permet de définir la décrémentation, via une paire de variables. Soit $\Phi : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \times \mathbb{N}$ la fonction

$$(m, n) \mapsto (n, n + 1).$$

En λ-calcul, on peut l'écrire comme :

$$\Phi := \lambda x.\texttt{2TUPLE}\ (\texttt{ELEM2}\ x)\ (\texttt{INC}\ (\texttt{ELEM2}\ x)).$$

On peut maintenant s'en servir pour décrémenter, soustraire, et tester si $m \geq n$:

$$\begin{aligned}
\texttt{DEC} &:= \lambda n.\texttt{ELEM1}(n\ \Phi\ (\texttt{2TUPLE}\ 0\ 0)) \\
\texttt{MOINS} &:= \lambda m\ n.n\ \texttt{DEC}\ m \\
\texttt{ÉGALEZÉRO} &:= \lambda n.n(\lambda x.\texttt{FAUX})\texttt{VRAI} \\
\texttt{PGE?} &:= \lambda m\ n.\texttt{ÉGALEZÉRO}\ (\texttt{MOINS}\ n\ m).
\end{aligned}$$

On peut aussi définir des fonctions qui s'appellent elles-mêmes :

$$\lambda f.(f\, f)$$

On peut utiliser tout ça pour calculer, par exemple, la suite de Fibonacci :

$$\texttt{G} := \lambda f.\lambda n.\texttt{SAS}\ (n = 0)\ 0$$
$$(\texttt{SAS}\ (n = 1)\ 1\ (\texttt{PLUS}\ (f\ f\ (n-1))\ (f\ f\ (n-2))))$$
$$\texttt{FIB} := \texttt{G}\ \texttt{G}.$$

Théorème de Church-Turing

Théorème 3.16

Les machines de Turing et le λ-calcul ont une puissance de calcul équivalente.

Pour prouver ce théorème, on doit démontrer comment ces deux modèles peuvent se simuler l'un l'autre.

λ-calcul $\rightarrow$ MT : Aperçu de la preuve

Pour simuler une MT avec le λ-calcul :

- Configuration (u, q, v) : $\texttt{3TUPLE}\ u\ q\ v$, où v est représenté par une liste, u par une liste en ordre inverse, et q simplement par un nombre naturel.
- δ : on définit une fonction qui prend une configuration en entrée et produit une configuration en sortie (ce qui peut devenir un peu compliqué...)
- Pour la boucle qui applique la fonction de transition δ, on utilise le même truc que pour Fibonacci :

$$\texttt{M}' := \lambda f.\lambda c.\texttt{SAS}\ \Big((\texttt{ELEM2}\ c) = q_A\Big)\ \texttt{VRAI}\ \Big(\texttt{SAS}\ ((\texttt{ELEM2}\ c) = q_R)\ \texttt{FAUX}\ (f\ f\ c)\Big)$$

et la vraie fonction est alors $\texttt{M} := \texttt{M}'\ \texttt{M}'$.

Le λ-calcul a servi de base à plusieurs langages de programmation :

- Haskell
- OCaml
- Lisp
- Scheme
- Miranda
- ...

Certains autres langages ont emprunté le « λ » du λ-calcul, dont Python :

```
carre = (lambda x: x*x)
```

Thèse de Church-Turing

Théorème de Church et Turing : « Les machines de Turing et le λ-calcul sont équivalents. »

Thèse de Church-Turing : « Tout ce qui est intuitivement calculable par un algorithme est calculable par une machine de Turing. »

« Tout modèle de calcul raisonnable peut être simulé efficacement avec une machine de Turing. »

Fonctions calculables

Définition 3.17

Les fonctions calculées par les MT sont appelées calculables.

Les fonctions calculables sont aussi appelées *fonctions récursives*.

Les booléens

Définition 3.18

Une *valeur booléenne* ou un *booléen* est un élément de l'ensemble $\{\langle \textsc{Vrai} \rangle, \langle \textsc{Faux} \rangle\}$.

On utilise aussi la convention que $\langle \textsc{Vrai} \rangle = 1$ et $\langle \textsc{Faux} \rangle = 0$, ce qui permet de représenter une valeur booléenne par un bit.

Définition 3.19

Pour un booléen x, $\neg x$, dit *non* x ou la *négation de* x, est un booléen qui est $\langle \textsc{Vrai} \rangle$ si et seulement si x est $\langle \textsc{Faux} \rangle$.

Définition 3.20

Pour x et y deux booléens, $x \wedge y$, dit x *et* y, ou la *conjonction de* x *et* y, ou le *produit booléen de* x *et* y, est un booléen qui est $\langle \textsc{Vrai} \rangle$ si et seulement si $x = \langle \textsc{Vrai} \rangle$ et $y = \langle \textsc{Vrai} \rangle$.

Définition 3.21

Pour x et y deux booléens, $x \vee y$, dit x *ou* y, ou la *disjonction de* x *et* y, ou la *somme booléenne de* x *et* y, est un booléen qui est $\langle \textsc{Vrai} \rangle$ si et seulement si $x = \langle \textsc{Vrai} \rangle$ ou $y = \langle \textsc{Vrai} \rangle$.

Expressions booléennes

Définition 3.22

Une *expression booléenne* est une expression contenant des variables, des valeurs booléennes et les opérateurs $\neg$, $\wedge$ et $\vee$.

Exemple 3.23

$$(x_1 \vee x_2) \wedge ((\neg x_1 \vee (x_3 \wedge x_2)) \wedge \neg x_3)$$

est une expression booléenne contenant les variables x_1, x_2 *et* x_3.

Si $x_1 = 1, x_2 = 1$ et $x_3 = 1$, alors la valeur de l'expression est

$$\begin{aligned}
(1 \vee 1) \wedge ((\neg 1 \vee (1 \wedge 1)) \wedge \neg 1) &= 1 \wedge ((\neg 1 \vee (1 \wedge 1)) \wedge \neg 1) \\
&= 1 \wedge ((0 \vee (1 \wedge 1)) \wedge \neg 1) \\
&= 1 \wedge ((0 \vee 1) \wedge \neg 1) \\
&= 1 \wedge (1 \wedge \neg 1) \\
&= 1 \wedge (1 \wedge 0) \\
&= 1 \wedge 0 \\
&= 0.
\end{aligned}$$

Circuits booléens

Définition 3.24 (Circuit booléen)

Soit $\mathcal{B}$ un ensemble fini de fonctions booléennes (qu'on appellera les *portes*). Un *circuit booléen* à n inputs et m outputs est un graphe dirigé acyclique contenant trois types de sommets :

- Les n *inputs*, étiquetés de 1 à n, qui n'ont pas d'arêtes entrantes
- Les *portes*, chacune étiquetée par un élément de $f \in \mathcal{B}$, dont le nombre d'arêtes entrantes est égal au nombre d'inputs de f
- Les m *outputs*, étiquetés de 1 à m, qui ont une seule arête entrante et aucune arête sortante.

De plus, pour chaque sommet, il existe un ordre sur les arêtes entrantes pour distinguer les différents inputs du f correspondant.

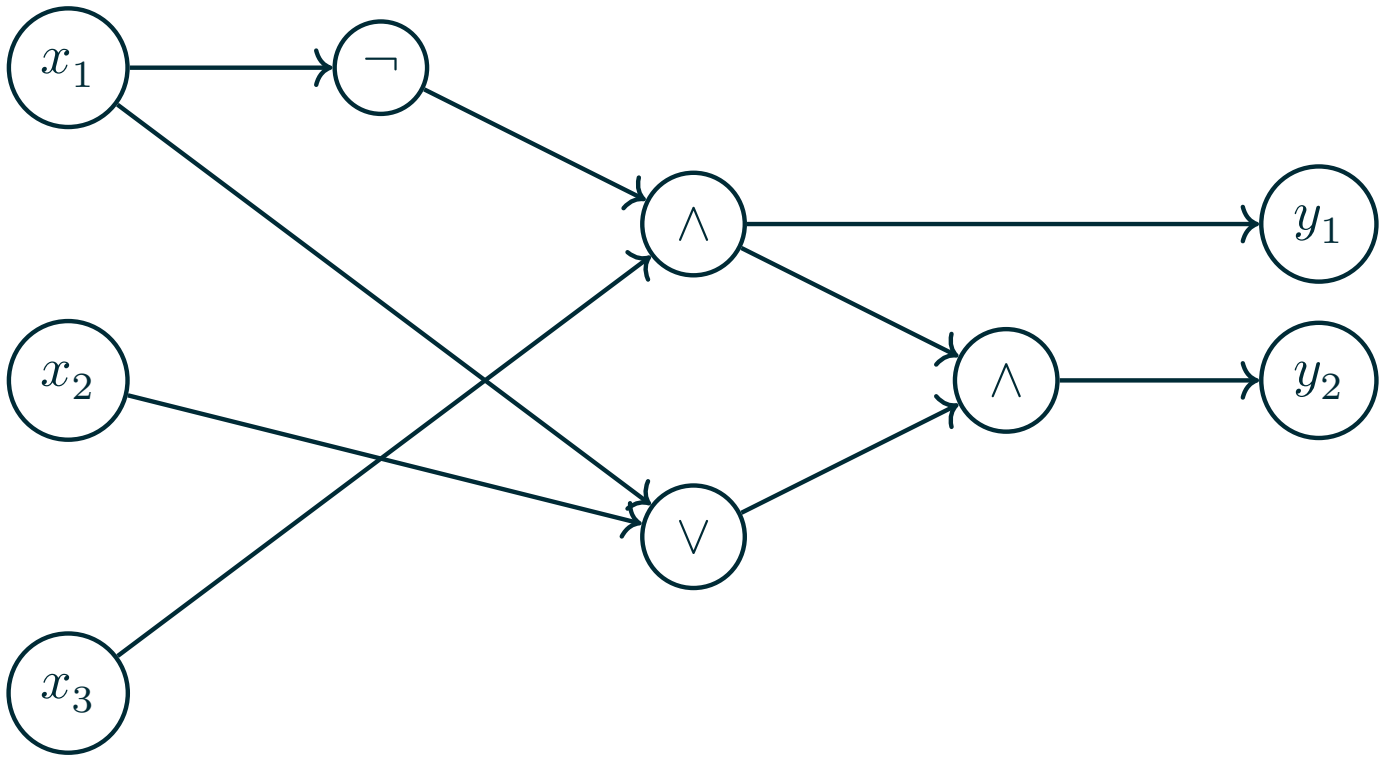
Définition 3.25

Étant donné un circuit booléen C à n inputs et m outputs, une *évaluation* de C sur l'input $x_1, \cdots, x_n$ est donné par un étiquetage des arêtes par des booléens tel que

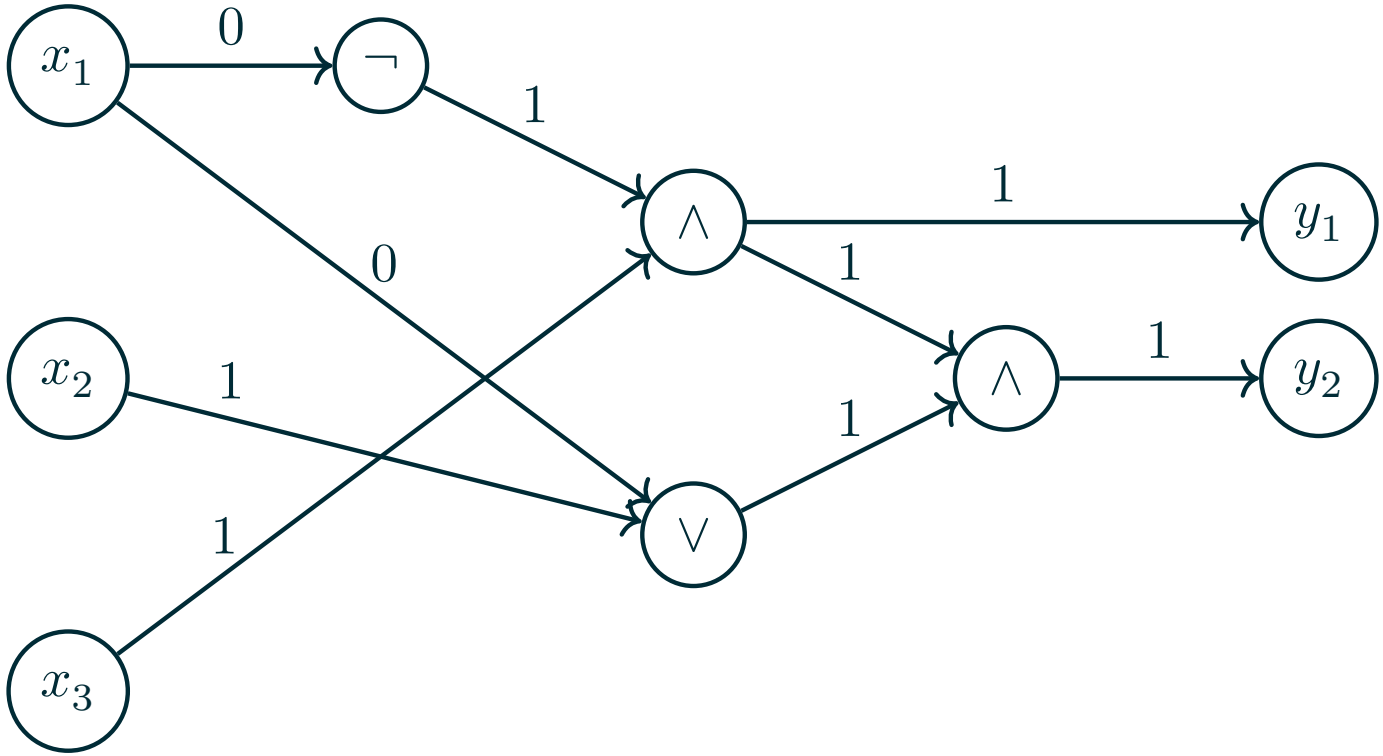
- les arêtes sortantes de l'input i sont étiquetées par le bit x_i
- pour chaque porte $f \in \mathcal{B}$, les arêtes sortantes sont étiquetées par f appliqué aux bits des arêtes entrantes.

L'output y_i du circuit est égal à l'étiquette de l'arête entrante du sommet correspondant à l'output i.

Voici un exemple de circuit booléen avec 3 inputs, 2 outputs, et $\mathcal{B} = \{\neg, \wedge, \vee\}$:



(Vu que toutes les portes sont symétriques, pas besoin d'ordonner les arêtes.)



Ce circuit donne :

$$y_1 = 1 \qquad y_2 = 1$$

si

$$x_1 = 0 \qquad x_2 = 1 \qquad x_3 = 1$$

Théorème 3.26

Toute fonction booléenne $f(x_1, \cdots, x_n)$ peut être représentée par une expression booléenne, et donc par un circuit booléen.

Démonstration :

Le circuit est donné par une disjonction de termes

$$t_1 \vee t_2 \vee \cdots \vee t_k.$$

où chaque t_i est une conjonction de toutes les variables, certaines variables pouvant être niées.

Pour chaque n-tuple $(a_1, \cdots, a_n)$ tel que $f(a_1, \cdots, a_n) = \langle \text{VRAI} \rangle$, on a exactement un terme. Ce terme prendra la valeur $\langle \text{VRAI} \rangle$ ssi $x_1 = a_1, \cdots, x_n = a_n$. □

Exemple 3.27

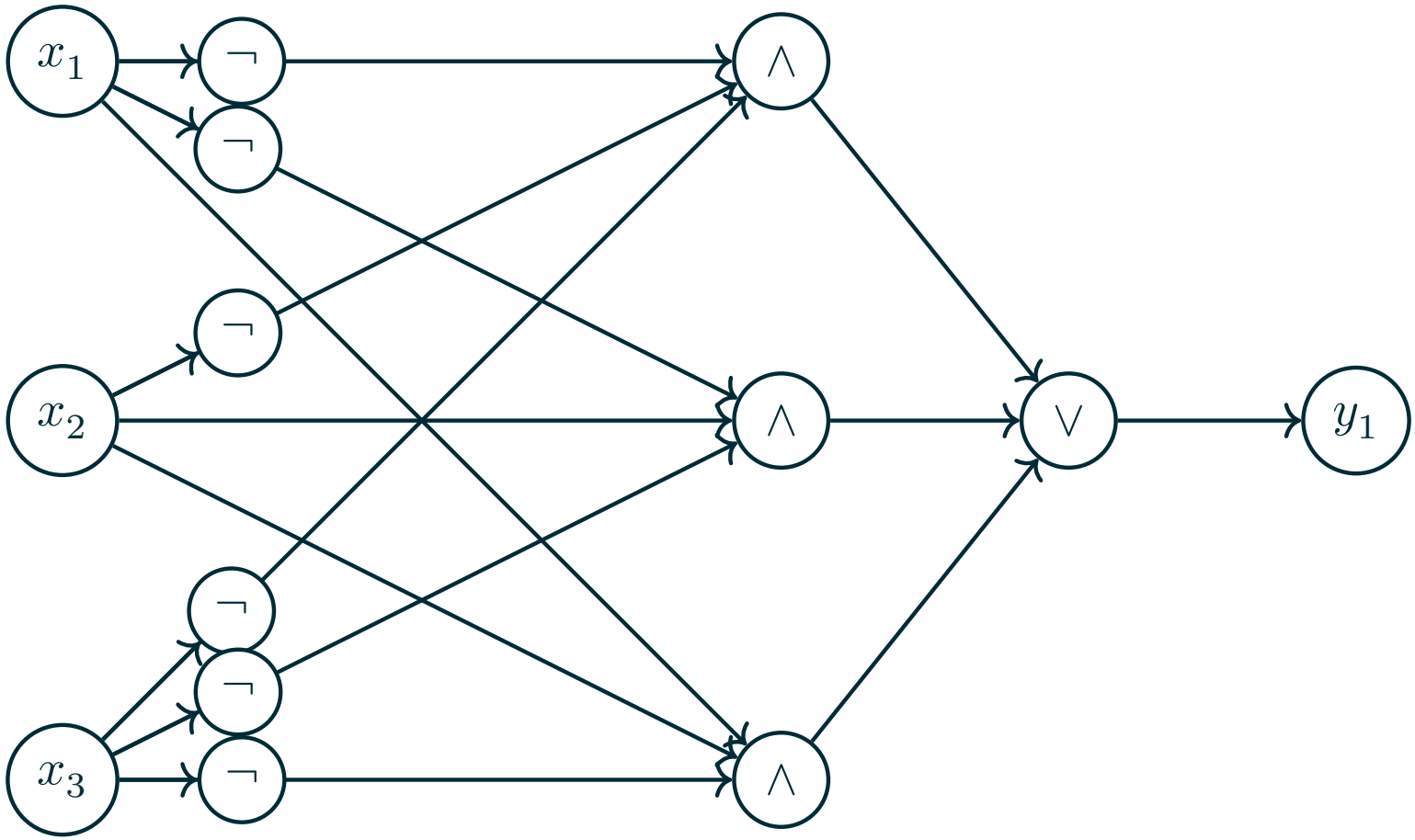
Soit la fonction f définie par le tableau suivant :

x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

alors l'expression booléenne correspondante est

$$(\neg x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (\neg x_1 \wedge x_2 \wedge \neg x_3) \vee (x_1 \wedge x_2 \wedge \neg x_3).$$

Le circuit booléen sur $\mathcal{B} = \{\neg, \wedge, \vee\}$ correspondant est alors :



$$y_1 = (\neg x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (\neg x_1 \wedge x_2 \wedge \neg x_3) \vee (x_1 \wedge x_2 \wedge \neg x_3).$$

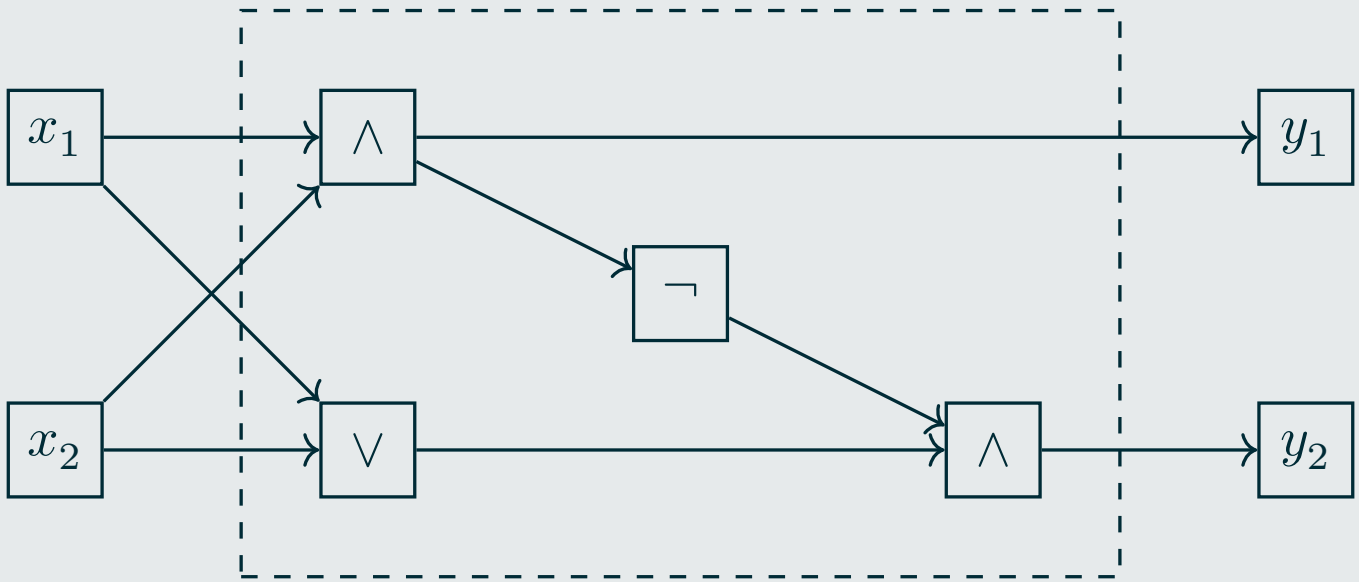
Remarque 3.28

Le théorème 3.26 peut être facilement généralisé aux fonctions de n variables booléennes à m variables booléennes :

$$(y_1, \cdots, y_m) = f(x_1, \cdots, x_n)$$

Exemple 3.29

Le circuit suivant calcule la somme de deux bits $+_2 : \{0,1\}^2 \to \{0,1\}^2$ *:*



Circuits booléens vs machines de Turing

Les circuits booléens sont-ils aussi puissants que les MT ?

- Techniquement, non : ils sont de taille fixe !
- Pour calculer une fonction sur un input de taille variable comme les MT, il faut une *famille* de circuits, avec un circuit pour chaque taille d'input.
- Famille non-uniforme : circuit arbitraire pour chaque n
- Famille uniforme : le $n$ième circuit est calculable...par une MT !