Mathématiques appliquées à l'informatique

Q1

Q1

Ch 3 : Arithmétique modulaire

$$ 349 = 349 * 1 $$

Les nombres premiers sont des nombres qui ne peuvent être divisé entièrement que par eux même et par 1. Par exemple 349 ne peut être divisé entièrement que par 349 et 1. Soit la seule factorisation possible de p est $p = 1 * p$. Exemples: 2, 3, 5, 7, 11, 13, 17, ...

$$ 1875 = 3 * 5 * 5 * 5 * 5 $$

N'importe quel nombre peut être factorisé en nombres premiers. Et il y a une infinité de nombres premiers.

Tester si un nombre est premier en utilisant un algorithme basique

```
def is_prime(n):
  for i in range(2,n):
    if (n%i) == 0:
      return False
  return True
```

Pour tester si un nombre est premier, on peut tester la division de tous les nombres jusqu'a $ \sqrt{n} $. Ce qui est peu efficace car on va aussi tester des nombres non premiers.

Tester si un nombre est premier en utilisant le crible d'Eratosthène

Pour calculer tous les nombres premiers $ \leq n $.

1. Lister les entiers de 2 à n

2. Pour chaque nombre plus petit que $ \sqrt{n} $ barrer tous les multiples du nombre (mais pas le nombre en lui même).
3. Tous les nombres qui restent sont des nombres premiers

Décomposer un nombre en produits de facteurs premiers

1. On divise le nombre successivement par des nombres premiers de plus en plus grand jusqu'a arriver à 1. Dans cette exemple, on va factoriser le nombre 4410.

$$ 4410 / 2 / 3 / 3 / 5 / 7 / 7 = 1 $$

2. Donc On arrive avec les valeurs suivante

$$ 4410 = 2 * 3 * 3 * 5 * 7 * 7 $$

3. Qui peut donc être simplifié de la manière suivante

$$ 4410 = 2 * 3^2 * 5 * 7^2 $$

Q1

La division euclidienne

Pour faire la division euclidienne d'un nombre je fais la division euclidienne des nombres positifs à la calculatrice.

N/D	Q	R
58/9	6	4
-58/9	$-6 - 1 = -7$	$-58 = 9 * -7 = 5$
58/-9	-6	4
-58/-9	7	5

Division euclidienne de deux polynomes

Q1

Le modulo

Trouver l'inverse modulaire

Prenons l'exemple de $9\mod80$

On peut écrire 9 et 80 dans le tableau suivant

R	9 (u)	80 (v)	Q
9	1	0	
80	0	1	

Ensuite on peut effectuer la division euclidienne des deux dernières lignes soit $9 \div 80$ et mettre le reste dans la première colonne et le quotient dans la dernière

R	9 (u)	80 (v)	Q
9	1	0	
80	0	1	
9			0

Maintenant on va multiplier chaque avant-dernière case moins chaque dernière case des deux dernières colonnes avec le quotient que l'on a trouvé

R	9 (u)	80 (v)	Q
9	1	0	
80	0	1	
9	$1-0*0=1$	$0-1*0=0$	0

Enfin on peut recommencer de nouveau depuis l'étape 2 jusqu'a arriver à un reste qui vaut 1. Si il n'y a pas de 1 et que l'on passe directement à 0, alors il n'y a pas d'inverse modulaire.

R	9 (u)	80 (v)	Q
9	1	0	

R	9 (u)	80 (v)	Q
80	0	1	
9	$1-0*0=1$	$0-1*0=0$	0
8	-8	1	8
1	$1-(-8)*1=9$	$0-1*1=-1$	1

Maintenant on peut prendre le résultat de la colonne u et :

- si celui-ci est plus grand que 80 on fait $80 - u$
- si celui-ci est plus petit que 0 on fait $80 + u$
- si celui-ci est entre les deux on le garde tel quel

Donc ici 9 est plus grand que 0 et plus petit que 80 donc le résultat est **9**

Faire le modulo d'un exposant négatif

Pour faire le $(690^{-6}) \mod 11$ on commence par faire l'inverse modulaire de 690 ce qui nous donne 7

Ensuite on fait $7^6 \mod 11$ ce qui nous donne donc 4 et c'est notre réponse finale.

Q1

Congruences

$$ 13 \mod 7 = 27 \mod 7 = 6 $$

> Deux nombres sont congrus si ils ont le même reste à la division euclidienne.

$$ a \mod n = b \mod n $$

Voici un énoncé plus court de la formule :

$$ a \equiv b (\mod n) $$

- Si a est congru à b alors $ a - b $ est divisible par n (peut être écrit comme $ a \equiv 0 (\mod n) $, 0 est le maximum donc si quelque chose n'est pas sous cette forme ce ne veut pas dire qu'il n'est pas divisible)
- Si a est congru à b et $ \alpha $ est congru à $ \beta $. Alors $ a+b $ et $ \alpha + \beta $ sont également congru, ainsi que $ a * b $ et $ \alpha * \beta $.

Q1

Ch 4 : Logique mathématique

La logique fournis des règles, des techniques permettant de décider si un raisonnement est valide ou pas.

La logique est utilisé en informatique, par exemple, pour définir les conditions qui détermineront la poursuite d'une boucle ou le choix d'une alternative au sein d'un programme, ou encore en SQL pour demander des choses à une base de donnée.

Q1

Propositions et prédicats

Proposition

Enoncé potentiel	Proposition ?	Raison	Valeur
Grand	Non	Trop ambigu	N/A
$ 4 = 9 $	Oui	N/A	Faux
8	Non	Aucune comparaison	N/A
"Je vais gagner au lotto"	Non	Pas de certitude, pas de connecteur logique	N/A
$ 7 + 9 > 11 $	Oui	N/A	Vrai

Une proposition est un énnoncé dont on peut dire avec certitude s'il il est vrai ou non. Une proposition n'est donc pas ambigue et peut avoir seulement 2 valeurs (1 (vrai) ou 0 (faux)).

Prédicats

Contrairement à une proposition un prédicat dépends d'une variable extérieure.

Q1

Connecteurs logiques de base

Minecraft logic gates

Image not found or type unknown

Sym	Nom mathématique	Electronique	Java	Minecraft
$ \neg $	Negation	`NOT`	`!`	Une torche de redstone sur un bloc avec un levier sur le bloc (inverseur)
$ \wedge $	Conjonction	`AND`	`&&`	2 torches activée par des leviers, reliées par une poudre de redstone et un inverseur
$ \vee $	Disjonction	`OR`	`||` (ou inclusif)	Une poudre de redstone qui relie 2 leviers
$ \oplus $	Disjonction exclusive	`XOR`	`^` (ou exclusif)	Trop compliqué à décrire
$ \iff $	Equivalence	`XNOR`	`==`	XOR avec un inverseur

La négation $ \neg $

La négation correspond à "Il est faut que P" (P étant une proposition)

Donc P prends la valeur contraire de $ \neg P $.

P	$ \neg P $	$ \neg \neg P $
0	1	0
1	0	1

Ceci est une table de vérité qui représente différentes valeurs d'énoncés par rapport à leur proposition(s).

La conjonction $ \wedge $

La conjonction est vraie si 2 propositions sont vraies.

P	Q	$ P \wedge Q $
0	0	0
1	0	0
0	1	0
1	1	1

La disjonction $ \vee $

La disjonction est vraie si une des 2 propositions sont vraie.

P	Q	$ P \vee Q $
0	0	0
1	0	1
0	1	1
1	1	1

Implication $ \implies $

$$ P \wedge Q \implies P \implies Q $$

Cet énoncé signifie que si P et Q sont vrai (AND) alors P est vrai et Q est vrai. L'implication correspond à "alors". Donc si XYZ alors ABC.

$$ \neg(x < y) \implies x = y $$

On peut par exemple imaginer une situation comme celle ci dessus, si x n'est pas plus petit que y alors x est égal à y.

P	Q	$ P \wedge Q $	$ P \implies Q $	$Q \implies P $
0	0	0	1	1
0	1	0	1	0
1	0	0	0	1
1	1	1	1	1

Contrairement aux conjonctions (AND), si la première proposition (l'antécédent) est faux, alors l'implication est forcément vraie car le contraire n'a pas été prouvé.

Donc les deux propositions ne sont pas interchangables comme vu dans le tableau les colonnes 4 et 5 ne sont pas les mêmes. BB Un peu de vocabulaire :

Antécédent	Conséquent	Enoncé	Réciproque	Contraposée
P	Q	$ P \implies Q $	$Q \implies P $	$\neg Q \implies \neg P $

Equivalence $ \iff $

P	Q	$P \iff Q $	$Q \iff P $
0	0	1	1
1	0	0	0
0	1	0	0
1	1	1	1

L'équivalence est une genre d'égalité dans la logique. Cela signifie que P corresponds à Q. Les deux propositions sont interchangables.

Disjonction exclusive $ \oplus $

$$ P \oplus Q \iff (P \wedge \neg Q) \vee (\neg P \wedge Q) \iff \neg (P \iff Q) $$

La disjonction exclusive peut etre représentée par des AND, OR et NOT uniquement, c'est un genre de raccourcis. La disjonction exclusive (XOR) peut aussi être utilisée pour représenter une équivalence en ajoutant une négation (NOT).

P	Q	$ P \oplus Q $	$ P \vee Q $	$P \wedge Q $
0	0	0	0	0
1	0	1	1	0

P	Q	$ P \oplus Q $	$ P \vee Q $	$P \wedge Q $
0	1	1	1	0
1	1	0	1	1

Une disjonction exclusive (XOR) est comme une disjonction inclusive (OR) sauf que les deux propositions ne peuvent pas être vrai pour que le XOR soit vrai. Donc un XOR sera forcément faux là ou un AND sera vrai.

Quantificateurs

En plus des connecteurs logiques on peut aussi ajouter les quantificateurs :

Symbole	Signification	Exemple
$\forall$	Pour tous	« $\forall x \in N : x * 2$ est paire » → *Pour chaque valeur x dans N quand elle est multipliée par deux est paire*
$\exists$	Existe	« $\exists x \in N : x$ est paire » → *Il existe au moins un nombre dans l'ensemble N qui est pair*

Q1

Formules en logique

$$ \neg ((P \vee Q) \wedge R) $$

La formule précédente est une combinaison de connecteurs. Les parenthèses sont très importante car comme en arithmétique, les parenthèses indiquent les priorités des opérations, donc $ P \vee Q $ doit être fait avant la plus grande parenthèse, pour enfin terminer par la négation complète.

Antilogies & Tautologie

$$ \vdash P \vee \neg P $$

La formule ci-dessus signifie que $ P \vee \neg P $ donnera **toujours** 1. Ca s'appelle une tautologie. Une tautologie signifie qu'une formule sera toujours vraie.

$$ \vdash \neg (P \wedge \neg P) $$

Tandis qu'une antilogie signifie qu'une formule sera toujours fausse.

Q1

Calcul booléen et table de Karnaugh

https://www.youtube-nocookie.com/embed/buM3XxdRxU0

https://www.youtube-nocookie.com/embed/4jndJ6ADiiE

Tableau de Karnaugh

Une première manière de représenter une fonction logique avec des opérateurs booléens est d'utiliser les formes normales tel que vue au cours d'architecture des ordinateurs.

Mais pour des fonctions plus complexes, les formes normales ne sont pas très efficace pour représenter de manière concise la fonction.

Former le tableau (code binaire réfléchit)

Tout d'abord on liste les différents paramètres de la fonction. Imaginons a,b,c. On va donc avoir par exemple a et b horizontalement et c verticalement :

c/ab	?	?	?	?
?				
?				

Ensiute pour chaque groupe, on va les écrire côte à côte

```
 a   b
--- ---
    0
    1
```

Ensuite pour ajouter le chiffre suivant on va faire un "mirroir" de ce que l'on a écrit

```
 a   b
--- ---
    0
    1
-------
    1
    0
```

Et sur la colonne d'a côté on va écrire des 0 sur la première partie, et des 1 sur la deuxième

```
 a   b
--- ---
 0   0
 0   1
-------
 1   1
 1   0
```

Ce qui nous donne donc le code binaire réfléchit, et ainsi les intitulé de notre colonne → `00, 01, 11, 10`

c/ab	00	01	11	10
0				
1				

On peut donc écrire dans chaque case les valeurs de la fonction correspondante. Par exemple la deuxième case signifie que a est à 0, b est à 1 et c est à 0. On peut ainsi compléter le tableau.

Comment transformer une forme normale en tableau de karnaugh

1er forme (disjonctive)

Pour la première forme on va dire que chaque groupe (produit) correspond à un 1

$$ F(a,b,c) = \overline{a} * \overline{b} * c + \overline{a} * b * c + a * \overline{b} * \overline{c} + a * \overline{b} * c + a * b * c $$

On peut ensuite construire le tableau de karnaugh :

c/ab	00	01	11	10
00				
01				

Pour chaque groupe on peut indiquer un 1 dans le tableau, par exemple $\overline{a} * \overline{b} * c$ c'est l'équivalent de dire `a=0`, `b=0` et `c=1` on peut donc l'indiquer dans la case correspondante du tableau :

c/ab	00	01	11	10
0				
1	1			

Et ainsi de suite pour le reste du tableau, ce qui nous donne :

c/ab	00	01	11	10
0				1
1	1	1	1	1

On complète ensuite les cases restantes par des 0

c/ab	00	01	11	10
0	0	0	0	1
1	1	1	1	1

2e forme (conjonctive)

Pour la deuxième forme normale c'est un peu plus bizarre. Disons que l'on part de la forme suivante :

$$ F(a,b,c) = (a+b+c) * (\overline{a}+b+c) * (\overline{a}+b+\overline{c}) * (\overline{a}+\overline{b}+c) $$

Ensuite on inverse tous les paramètres

$$ F(a,b,c) = (\overline{a}+\overline{b}+\overline{c}) * (a+\overline{b}+\overline{c}) * (a+\overline{b}+c) * (a+b+\overline{c}) $$

Ensuite on procède exactement comme avant sauf qu'a la place d'écrire des 1 on écrit des 0.

c/ab	00	01	11	10
00	0		0	0

c/ab	00	01	11	10
01				0

Enfin on complète le reste avec des 1 :

c/ab	00	01	11	10
00	0	1	0	0
01	1	1	1	0

Simplifier un tableau de Karnaugh

Maintenant que l'on sait comment créer un tableau de Karnaugh, on va maintenant l'utiliser pour obtenir une forme plus abrégée des fonctions.

cd/ab	00	01	11	10
00	1	0	1	1
01	1	0	1	0
11	1	0	0	1
10	1	0	0	1

On peut ensuite grouper les différents éléments par puissance de 2 (donc par 1, 2, 4, 8, etc). **ATTENTION**, il faut imaginer le tableau comme une boule, les 1 sur les extrémités peuvent être reliées entre elles. Donc si par exemple il y a un 1 dans chaque coin, ils peuvent être connecté comme un seul groupe.

Par exemple voici un groupe (un peu complexe) de 1 dans tous les coins :

cd/ab	00	01	11	10
00	**1**	0	1	**1**
01	1	0	1	0
11	1	0	0	1
10	**1**	0	0	**1**

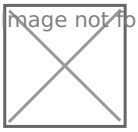
Ensuite on peut regarder quels sont les paramètres qui ne varient pas :

cd/ab	00	01	11	10
0**0**	**1**	0	1	**1**
01	1	0	1	0

cd/ab	00	01	11	10
11	1	0	0	1
1**0**	**1**	0	0	**1**

On remarque donc que dans tous les cas du groupe, *a* et *b* sont à 0. Donc l'expression en produit de notre groupe est $\overline{b} * \overline{d}$

On peut ensuite faire la même chose pour tous les groupes ce qui nous donne :



Il faut donc essayer de toujours prendre les plus grand groupes (pour avoir les plus petits produits) et d'en prendre le moins possibles (pour avoir le moins de produits possibles).

Cela va donc nous donner l'expression simplifiée suivante :

$$ F(a,b,c) = \overline{b} * \overline{d} + \overline{a} * \overline{b} + \overline{c} * a * b + \overline{b} * c $$

Q1

Les ensembles

Un ensemble est une collection non-ambigue d'objet distincts. C'est à dire que l'on peut définir ce qui relie tous les objets, et que les objets ne peuvent apparaitre qu'une seule fois dans l'ensemble.

Voici quelques exemples d'ensembles :

- Des ensembles finis de nombres : $A = \{ 0, 1, 2, 5, 9, 11 \}$
- Des ensembles finis de noms : $B = \{ Alain Dupont, Béatrice Durant, Linel Hicq, Nadine Tudor \}$
- Des ensembles infinis : $C = \{1,2,3,4,5,...\}$

On défini un ensemble par la caractéristique commune à tous les éléments

$$ A = \{x | x \in \mathbb{N}\} $$

Cette notation fait appel à la notion de préciats que l'on a vu plus tôt, voir ici, car on a un prédicat sur la variable x

$$ A = \{ x | P(x)\} $$

Quand un ensemble ne contient aucun élément on dit que c'est un ensemble "vide"

$$ B = \{\varnothing\} $$

Cardinalité des ensembles

Pour connaitre le cardinal d'un ensemble, il suffit de compter ses éléments.

- Si c'est un ensemble vide ($\varnothing$), alors le cardinal est 0
- Si c'est un ensemble qui contient d'autres ensembles, on ne fait que compter les ensembles (sans leur contenu)

Ainsi pour l'ensemble suivant :

$$ A = \{\{A\}, \{A,C\}, B, \{B,C,D,E\}, D, \{D,E\},H\} $$

Cet ensemble a un cardinal de 7.

Les relations entre les ensembles

- L'égalité. C'est à dire que a appartient à b et b appartient à a. → $\forall x \in E, (x \in A) \iff (x \in B)$

En plus de l'égalité on a aussi les opérations ensemblistes : **Attention** comme dans tous les ensembles il n'y a pas besoin de répèter les nombres.

Nom	Expression mathématique	Description
L'union	$A \cup B = \{x\|x \in A \lor x \in B\}$	soit tous les éléments qui sont dans A ou qui sont dans B
L'intersection	$A \cap B = \{x\|x \in A \land x \in B \}$	soit tous les éléments qui sont dans A et qui sont dans B
La différence	$A \setminus B$ ou $A - B = \{x\|x \in A \land x \notin B \}$	tous les éléments qui sont dans A mais pas dans B
La différence symétrique	$A \oplus B = \{ x \| (x \in A \land x \notin B) \cup (x \notin A \land x \in B) \}$	Tous les éléments qui sont uniquement dans A + tous les éléments qui sont uniquement dans B

Résoundre les diagrammes de Eulen-Venn

Pour arriver à trouver un les éléments d'un diagramme qui correspondent à une expression ensembliste, j'essaye de trouver des patterns dans l'expression.

Voici les patterns que j'ai identifiés :

- $A\cup B$ ou $A\setminus\overline{B}$ → Tous les éléments de A et tous les éléments de B (en faisant attention à ne pas répèter un même élément)
- $A\cap B$ → Les éléments communs à A et B
- $A\setminus B$ ou $A\cap\overline{B}$ → On prends les éléments de A et on retire ceux qui sont dans B
- $\overline{A}\setminus \overline{B}$ ou $\overline{A}\cap B$ → On prends les éléments de B et on retire ceux qui sont dans A
- $A\cup\overline{B}$ → Tous les éléments de B + tous les éléments qui ne sont pas dans A (en faisant attention à ne pas répèter plusieurs fois un même élément)
- $\overline{A}\setminus {B}$ ou $\overline{A} \cap \overline{B}$ → on prends tout sauf A et B

Récurrence et récursivité

Récurrence

Les suites

Une suite est une liste **ordonnée** d'éléments appelés "**terme**".

La **longueur** d'une suite correspond au nombre d'éléments qu'elle contient. Il existe quelques noms spécifiques pour certaines suites :

- **vide** désigne une suite composée de 0 éléments et est représentée ainsi : $()$
- **couple** désigne une suite composée de 2 éléments (exemple $(1,2)$)
- **finie** désigne une suite composée d'un nombre déterminé d'éléments (exemple *Les nombres entre 1 (inclus) et 3 (inclus)* : $(1,2,3)$)
- **infinie** désigne une suite composée d'un nombre indéterminé d'éléments (exemple "La suite des nombres positifs partant de 0 jusqu'à l'infini, ou la suite de Fibonacci" qui peuvent être représentée par une formule mathématique ou par une suite terminée par ... $(1,2,3,..., n)$)

Une suite (qui correspond plus aux collections `List` en Java) est notée avec des parenthèses et ne doit pas être confondue avec un **ensemble** qui est représenté par des accolades (et qui correspond aux collections `Set` en Java)

Une suite peut être transformée en ensemble en retirant tous les doublons qu'elle contient.

Ainsi la suite $(1,2,3,4,3,2)$ deviendra l'ensemble ${1,2,3,4}$. À savoir que contrairement aux suites, l'ordre ne compte pas dans un ensemble. Ce qu'il peut exister un nombre infini de suites pour un ensemble donné. L'ensemble correspondant à une suite est noté $E(s)$

> Note: La suite de fibionacci est une suite commençant par `0,1` où chaque élément est la somme des 2 précédents.

La démonstration par récurrence

Le but de la récurrence est de montrer qu'une propriété $P(n)$ est vraie pour tout n.

- Premièrement, l'**initialisation**, qui correspond à la preuve d'une propriété au premier rang demandé. Si on prend l'exemple de domino, l'initialisation est le fait que le premier domino tombe.
- Deuxièmement, l'**hérédité** qui correspond au fait que chaque élément se comportera toujours exactement comme le premier. Autrement dit, que si la propriété est valable pour n alors, elle sera aussi valable pour $n+1$. Ainsi, dans l'exemple du domino, l'hérédité est le fait que chaque domino qui tombe fera tomber son suivant.

> TODO démonstration par récurrence héréditaire et emboitée

L'intérêt de la récurrence en informatique, c'est le fait de pouvoir prouver mathématiquement le fonctionnement d'un programme ainsi diminuant la nécessité de faire un plan de test, car une démonstration vérifie que le programme fournit un résultat correct dans tous les cas de figures.

La récursivité

La récursivité est une implémentation dans un programme de la notion mathématique de récurrence. Ainsi la récursivité est *la démarche de faire référence à l'objet même de la démarche à un moment du processus* (par exemple une méthode qui s'appelle elle-même sous certaines conditions)

L'approche récursive est une manière de programmer qui s'oppose en quelque sorte à l'approche itérative. L'approche dite "itérative" va utiliser des boucles tandis que l'approche récursive va résoudre un problème en calculant les instances plus petites du même problème.

La récursivité est un moyen simple et élégant de résoudre un problème qui a également l'avantage d'être souvent plus simple et rapide à programmer. En revanche, elle a généralement le désavantage d'être plus gourmande en ressources pour l'ordinateur et ainsi d'être moins efficace.

Voici un exemple de définition d'un problème de manière *explicite*, *itérative* et *récursive* :

Considérons la suite des sommes des □□ premiers nombres entiers strictement positifs

- Explicite : $S_n = \frac{n(n+1)}{2}$
- Itérative : $S_n = \sum_{i=1} ^{n} i$
- Récursive : $S_n = S_{n-1} + n$

Ainsi, choisir la récursivité à la place d'une autre approche dépends de son goût personnel pour un style de programmation, de la simplicité de la solution récursive par rapport à une autre, et de l'efficacité recherchée de l'implémentation.

Créer une méthode récursive

Une méthode récursive va être composée de 2 choses au minimum

1. Une condition d'arrêt, c'est-à-dire un cas de base où on sait avec certitude la réponse.
2. Un appel à la fonction elle-même en décrémentant un de ses paramètres.

Mais on peut éventuellement lui ajouter une condition d'erreur si un paramètre est incorrect

Ainsi voici un calcul de la suite de Fibonacci *récursive*

```
public static int fibonacci(int n) {
    // 0. condition d'erreur
    if (n <= 0) throw new IllegalArgumentException("n doit être strictement supérieur à 0");

    // 1. condition d'arrêt de la récursion, on sait que quand n est égal à 2 ou a 1 la réponse sera n-1
    if (n <= 2) return n-1;

    // 2. appel récursif à la méthode en la décrémentant son paramètre n
    return fibonacci(n-1) + fibonacci(n-2);
}
```

Définition par induction

Il est possible de définir certaines notions par *induction*, c'est-à-dire en procédant en 2 temps :

1. Base initiale ou base de construction : on représente les éléments primitifs de la notion
2. Étape inductive, c'est-à-dire que l'on énonce les règles de la notion à partir des éléments primitifs

Par exemple pour représenter l'ensemble des nombres impairs (I)

1. Base initiale : $1 \in I$ (1 est l'élément primitif, car c'est l'instance la plus petite d'un nombre impair)
2. Règle de construction : $\forall n \in I : (n+2) \in I$ (pour tout nombre impair, ce même nombre + 2 sera aussi impair)

Voici un autre exemple avec un palindrome (séquences de lettres pouvant se lire de la même façon de droite à gauche et de gauche à droite)

1. Base initiale : Toute lettre x est un palindrome. Une lettre étant l'instance la plus petite d'un palindrome (une lettre est la même se lit de la même manière de droite à gauche et de gauche à droite)
2. Règle de construction : Une lettre x + un palindrome + lettre x est aussi un palindrome (par exemple `r`, `d` et `a` sont des palindromes, `ada` est un palindrome et `radar` est également un palindrome)

Cela est un peu équivalent à la manière de construire une méthode récursive en soi. La base initiale étant la condition d'arrêt et la règle de construction étant l'appel récursif.

Preuve de correction d'un algorithme récursif par induction

La méthodologie est la suivante :

1. Définir quel est le but de l'algorithme et une instance du problème
2. Instances ordonnées par taille (entier n, taille du tableau, etc)
3. Indiquer la base de l'induction comme vu plus tôt
4. Indiquer la règle de construction comme vu plus tôt également
5. Terminaison : on montre que les appels récursifs se font sur des sous-problèmes

Les langages formels

La théorie des langages formels fut initialement initiée par les linguistes qui tentaient de représenter les langages naturels (exemple, français, anglais, etc). Mais elle s'est révélée être très utile pour l'informatique pour tout un cas d'applications. Notamment les RegEx, et les règles syntaxiques des langages de programmations (avec l'analyse syntaxique).

Un langage naturel va être défini par une orthographe, une grammaire, une conjugaison. Un langage dit formel (tel que les langages de programmations), vont être défini par des **règles de constructions**. Ainsi, les erreurs syntaxiques d'un code peuvent être détectées grâce à ces mêmes règles de constructions.

La théorie des langages a pour but de décrire les langages formels. Elle ne concerne que les aspects purement syntaxiques du langage. Un langage formel est généralement défini par une grammaire formelle (grammaires algébriques, par exemple) et analysé à l'aide d'automates.

Donc un langage formel va être généré à l'aide d'une **grammaire formelle** et analysé à l'aide d'un **automate**.

Alphabets, lettres et mots

L'**alphabet** est un ensemble fini et non-vide d'éléments appelé lettres ou symboles terminaux. Cet ensemble va être représenté par la lettre Σ, par exemple : $\Sigma = {a,b,c,1,2,3,if,else}$

- Σ^+ correspond à l'ensemble de tous les mots sur Σ
- $\Sigma^* $ correspond à l'ensemble de tous les mots *non-vide* sur Σ

Un **mot** est une séquence finie de lettres. Ainsi dans l'alphabet précédent `a`, `ab` et `else` sont des mots par exemples.

Le **mot vide** est noté ε et est défini pour tous les alphabets. Par conséquent, le mot vide ne peut pas être une lettre de l'alphabet Σ.

Opérations

La concaténation est une opération binaire effectuée sur l'ensemble Σ et qui donne un mot obtenu par la concaténation des 2 mots. Elle est associative et non commutative et admet le mot vide ε comme neutre (donc si u est un mot alors $u = \varepsilon u = u

\varepsilon$).

Une concaténation des mots u et v est représentée par uv ou $u * v$ et une concaténation d'un nombre k de u est représentée par u^k donc $uuuuuuuuu = u^9$

Un exposant $+$ correspond à "1 ou plus" (comme en RegEx) tandis qu'un exposant $* $ sur un mot représente "0 ou plus" (toujours comme en regex)

Ainsi $(ab)^+$ signifie que la séquence `ab` est présente une fois ou plus. Et $(vb)^* $ signifie que la séquence `vb` est présente 0 fois ou plus.

Une autre opération est qu'un mot peut être retourné en lui mettant un exposant de `-1`. Donc $(abc)^{-1} = cba$ et un palindrome peut être détecté lorsque $u = u^{-1}$ (par exemple $(radar)^{-1} = (radar)$)

Et on peut aussi récupérer la longueur d'un mot (nombre de symboles terminaux/lettres) comme ceci $|u|$ donc $|(radar)| = 5$

Langage

Un **langage** est un sous ensemble de $\Sigma^* $ et on peut reconnaitre quelques langages particuliers :

- Σ est le langage de tous les mots de longueur 1 sur Σ
- $\varnothing$ est un langage ne contenant aucun mot
- {$\varepsilon $} est un langage ne contenant que le mot vide
- Σ^+ langage contenant tous les mots sur Σ^+
- $\Sigma^* $ langage contenant tous les mots sur $\Sigma^* $

Sur l'ensemble $P(\Sigma^*)$ représentant tous les langages sur $\Sigma^* $ on peut utiliser les opérations ensemblistes habituelles, mais également $\cdot$ permetant de faire un **produit** des deux langages en utilisant la concaténation sur chaque mot.

Aisni si $L_1 = 1,2,3$ et $L_2 = a,b,c$ et que l'on fait $L_1 \cdot L_2$ on obtient $1a,1b,1c,2a,2b,2c,3a,3b,3c$

On peut utiliser la notation L^k où k est le nombre de lettres, pour représenter une concaténation de k mots du langage L. Et $L^k = L \cdot L^{k-1}$ ainsi :

- $L^0 = \varepsilon$
- $L^1 = L \cdot L^{0} = L \varepsilon = L$
- $L^2 = L \cdot L^{1} = LL$

Pour décrire un langage avec une infinité de mots, on peut utiliser un système générateur (grammaire, RegEx ou diagramme syntaxique) pour générer des mots du langage et des automates finis pour les détecter.

Pour la concaténation de 2 langages, les exposants $+$ (1 ou plus de fois concaténé avec lui-même) et $* $ (0 ou plus de fois concaténé avec lui-même) restent valides, ainsi étant donnés les deux langages :

- $L_1 = ab,ac,c,cb,\varepsilon$
- $L_2 = a,ab,ca,cb,cc$

Quand on applique $(L_1 \cdot L_2)^* $ on peut obtenir des mots tel que `acbcb` car cela correspond à $(L_1 \cdot L_2)^2 = L_1 \cdot L_2 \cdot L_1 \cdot L_2 = \varepsilon \cdot a \cdot cb \cdot cb = acbcb$. Il faut donc bien penser à tous les mots dans toutes les combinaisons possibles. Mais quand on a un QCM avec plusieurs propositions, quand aucun mot ne commence avec certaines lettres, on peut les éliminer avec certitude, ce qui rend les choses un peu plus simples.

Grammaire

Ici, on va examiner les grammaires dites "de Chomsky" qui représente la forme la plus connue de *système de génération* de langages, qui sont définies par le quadruplet $G = (\Sigma, N,P,S)$ où les symboles signifient :

- G est la grammaire
- Σ est l'ensemble des symboles terminaux (par exemple : Marc, livre, Alice, mange, lit, pomme, une, un)
- N est l'ensemble des symboles non-terminaux (par exemple : sujet, verbe, complément, article, nom)
- P est l'ensemble des productions de la grammaire (c'est-à-dire les associations des symboles non-terminaux avec les symboles terminaux) (exemple, voir la capture d'écran d'une slide ci-dessous). On va voir plus tard que les productions sont des règles de remplacement.
- S est le symbole initial qui entame la génération de la grammaire. Par convention, ce sera S justement

8.2 – Grammaires

Exemple

- $\Sigma = \{\textbf{Marc}, \textbf{Alice}, \textbf{mange}, \textbf{lit}, \textbf{un}, \textbf{une}, \textbf{pomme}, \textbf{livre}\}$
- $N = \{S, sujet, verbe, complément, article, nom\}$
- S
- $P = \{ S \rightarrow sujet\ verbe\ complément$
  $complément \rightarrow article\ nom$
  $sujet \rightarrow \textbf{Marc}|\textbf{Alice}$
  $verbe \rightarrow \textbf{mange}|\textbf{lit}$
  $article \rightarrow \textbf{un}|\textbf{une}$
  $nom \rightarrow \textbf{pomme}|\textbf{livre}\ \}$

HEL MO Haute Ecole Libre Mosane TECHNIQUE

Raccourci d'écriture : $\alpha \rightarrow \beta_1, \ldots, \alpha \rightarrow \beta_n$ s'écrit $\alpha \rightarrow \beta_1 | \ldots | \beta_n$

Lorsque plusieurs règles de grammaire ont une même forme en partie gauche, on pourra factoriser ces différentes règles en séparant les parties droites par des traits verticaux

CH08.41

Conventions

- Par convention, les symboles en lettres latines minuscules représentent des symboles terminaux
- Les symboles latins majuscules représentent des symboles non-terminaux
- Les symboles en lettres grecs minuscules représentent des **suites** de symboles quelconques (terminaux ou pas)
- Les lettres latines minuscules allant de u à z sont considérées comme des **suites** de symboles terminaux.
- Les lettres latines majuscules allant de U à Z sont considérées comme des symboles quelconques

Dérivation

L'opération de base qui est effectuée dans une grammaire est la *dérivation*, autrement dit le fait de convertir des symboles non-terminaux en symboles terminaux sur base des règles de production de la grammaire. Cette opération est représentée par le symbole $\Rightarrow$

Ainsi si la grammaire indique que $A \Rightarrow \alpha$ et $S \Rightarrow abcA$ alors la génération de départ donnera $abc\alpha$ car A sera remplacé par α

Le langage qui sera généré d'une grammaire est noté $L(G)$ et correspond à l'ensemble Σ que l'on peut dériver à partir de la grammaire G

Il y a 2 types de dérivation, la dérivation *à gauche* et la dérivation *à droite*, autrement dit que l'on va d'abord s'occuper du terme le plus à gauche ou du terme le plus à droite. Voici un exemple venant des tests :

Soit une grammaire $G = (\Sigma, N, P, E)$ qui définit des expressions arithmétiques.

- $\Sigma = \{+, -, *, /, (,), n\} \rightarrow$ symboles terminaux (n représente un nombre)
- $N = \{E, T, F\} \rightarrow$ symboles non terminaux *(**E**xpression, **T**erme, **F**acteur)*
- $E \rightarrow$ symbole initial (axiome)
- **Productions**

$$E \rightarrow E + T | E - T | T$$
$$T \rightarrow T * F | T / F | F$$
$$F \rightarrow n | (E)$$

Donnez une **dérivation à gauche** de l'expression suivante : n*(n)/n
Note: les éventuelles cases inutiles doivent rester vides.

E

=> T

=> T/F

=> T*F/F

=> F*F/F

=> n*F/F

=> n*(E)/F

=> n*(T)/F

=> n*(F)/F

=> n*(n)/F

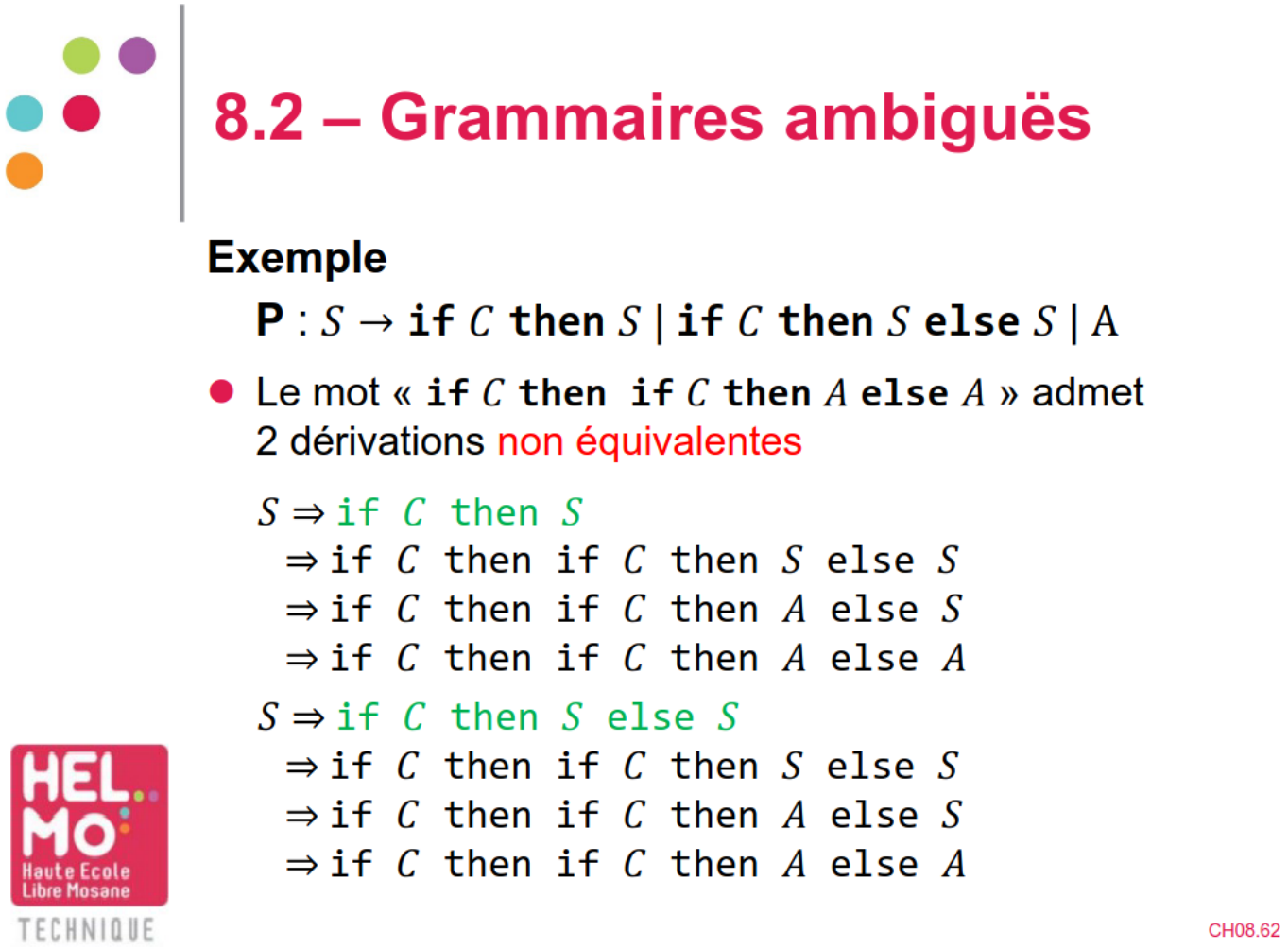
=> n*(n)/n

À savoir que la barre verticale dans ce cas veut dire "ou". Donc la première production veut dire "E peut devenir E+T ou E-T ou T" par exemple.

Dans une dérivation à droite, on aurait à la place toujours commencée par dériver le symbole non terminal le plus à droite.

Grammaire ambigüe

Une grammaire est dite **ambigüe** si un mot peut être généré par 2 dérivations de gauche non équivalentes. Voici un exemple de grammaire ambigüe :



Donc pour enlever l'ambiguïté, il faut forcer la dérivation dans un certain sens, par exemple avec les nouvelles règles de productions suivante :

□□ → if □□ then □□ | if □□ then □□ else □□ | A

□□ → if □□ then □□ else □□ | A

Tout langage généré par une grammaire ambigüe est dit **intrinsèquement ambigu** et ce type de langage est proscrit en informatique.

La dénomination de **langage déterministe** s'applique à tout langage issu de grammaire non ambigüe.

Les types de grammaires

Il existe différents types de grammaires en fonction de la complexité de leurs règles de productions :

- Type 0 (sans restriction) : les grammaires telles que $\alpha \Rightarrow \beta$ (soit une suite donne une autre)
- Type 1 (contextuelle) : les grammaires telles que $\alpha A\gamma \Rightarrow \alpha\beta\gamma$ (soit un symbole non-terminal est entouré d'au moins 2 mots qui donne le contexte. Le remplacement du symbole terminal va dépendre du contexte dans lequel il se trouve
- Type 2 (algébrique ou non contextuelle) : les grammaires telles que $A \Rightarrow \gamma$ sont la même chose que le type 1 sauf que le contexte est vide
- Type 3 (régulières) : les grammaires telles que $S \Rightarrow aA$ (alors appelée *linéaire à droite*) ou $S \Rightarrow Aa$ (alors appelée *linéaire à gauche*). Où le symbole non-terminal d'une règle est toujours placé d'un côté ou de l'autre des symboles non-terminaux. Ce type de grammaire engendre des **langages rationnels**, c'est-à-dire un langage qui peut être reconnu par un automate fini

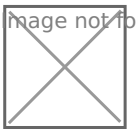
Automates finis

> Pour comprendre cette section, comprendre ce qu'est un diagramme d'état et ce que sont les langages formels est très utile.

Un automate fini est une construction mathématique abstraite représentant une machine ayant plusieurs états et des transitions permettant de passer d'un état à l'autre.

Les mécanismes

Les **mécanismes** sont des machines simples, **sans inputs ni output** et sans lien avec le monde extérieur. Elles ne sont dirigées **que par leur propre horloge interne**. En voici un exemple pour un feu tricolore :



Ils peuvent être décrits par le couple $M = (E, t)$ où E est l'ensemble des états et t la fonction de transition entre les états. Un premier état sera représenté par e et à l'instant suivant passera dans l'état $t(e)$.

Les mécanismes seront toujours une boucle infinie, il n'y a pas de point de départ ni de point d'arrivée. Et si un état n'a des transitions qu'avec lui-même ou à personne, alors on dira que c'est un **état repos**.

etat repos

Automates fini

Les automates finis sont un peu plus complexes que les mécanismes, car ils ont des inputs. Un automate fini est décrit par le triplet $A = (E, \Sigma, t)$ où E est l'ensemble fini et non-vide des états, Σ l'ensemble fini des inputs et t la fonction de transition.

Un état e après l'ajout d'un input i deviendra $t(e,i)$. Un diagramme d'état d'un automate fini sera représenter comme ceci :

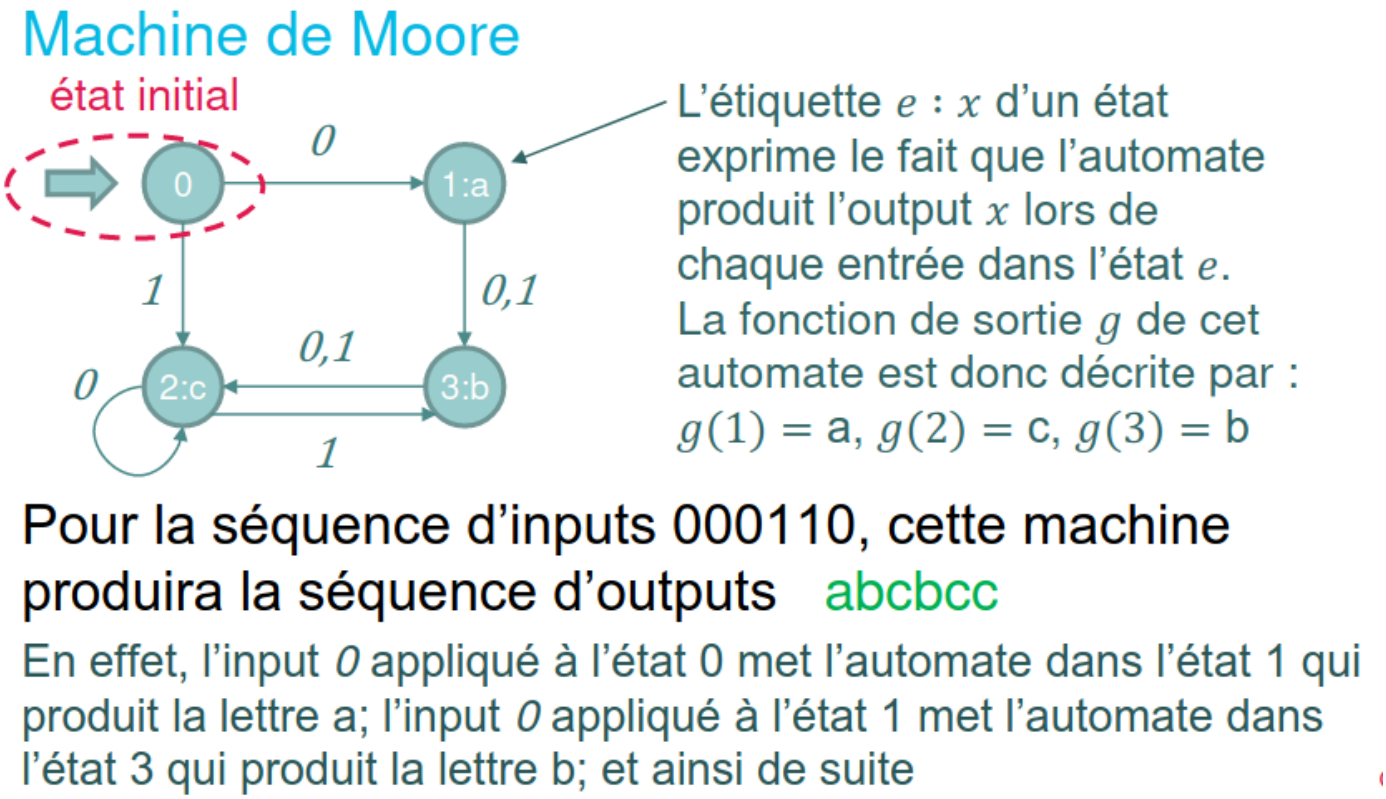
Image not found or type unknown

Machines de Moore et machines de Mealy

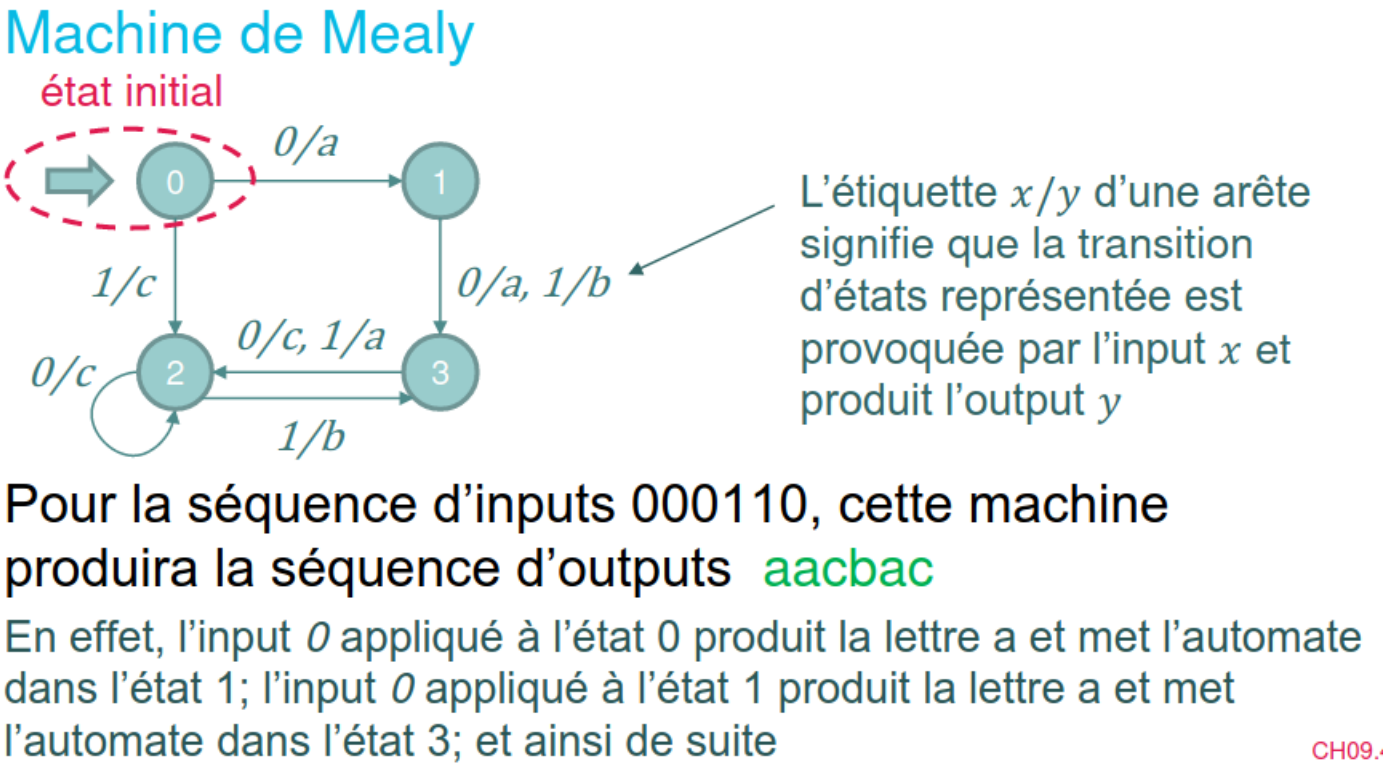
Les automates précédents ne servent pas à grand-chose, car ils ne produisent rien. Contrairement aux automates précédents, ceux-ci ont un état initial et une séquence d'output.

Les machines de Moore et de Mealy sont différentiées par ce qui détermine l'output :

- Dans une **machine de Moore**, la sortie est déterminée par l'état après la transition



- Dans une **machine de Mealy**, la sortie est déterminée par la transition elle-même



Les machines de Moore et Mealy sont des machines **séquentielles** qui transforment une suite (input) en une autre suite (outputs). Une machine de Moore est toujours un cas particulier de machine de Mealy et inversement une machine de Mealy a toujours une machine de Moore équivalente.

Machines de reconnaissance

Maintenant, on va s'intéresser à un type particulier de machine de Moore, les machines de reconnaissances. Elles ont comme seuls outputs 0 et 1. 0 si l'input n'est pas reconnu et 1 s'il l'est.

Par convention, un état non reconnu sera un simple rond, et un état reconnu (aussi appelé état d'acceptation) sera un double rond.

Cette machine va donc servir à vérifier si une séquence donnée mène ou non à un état d'acceptation.

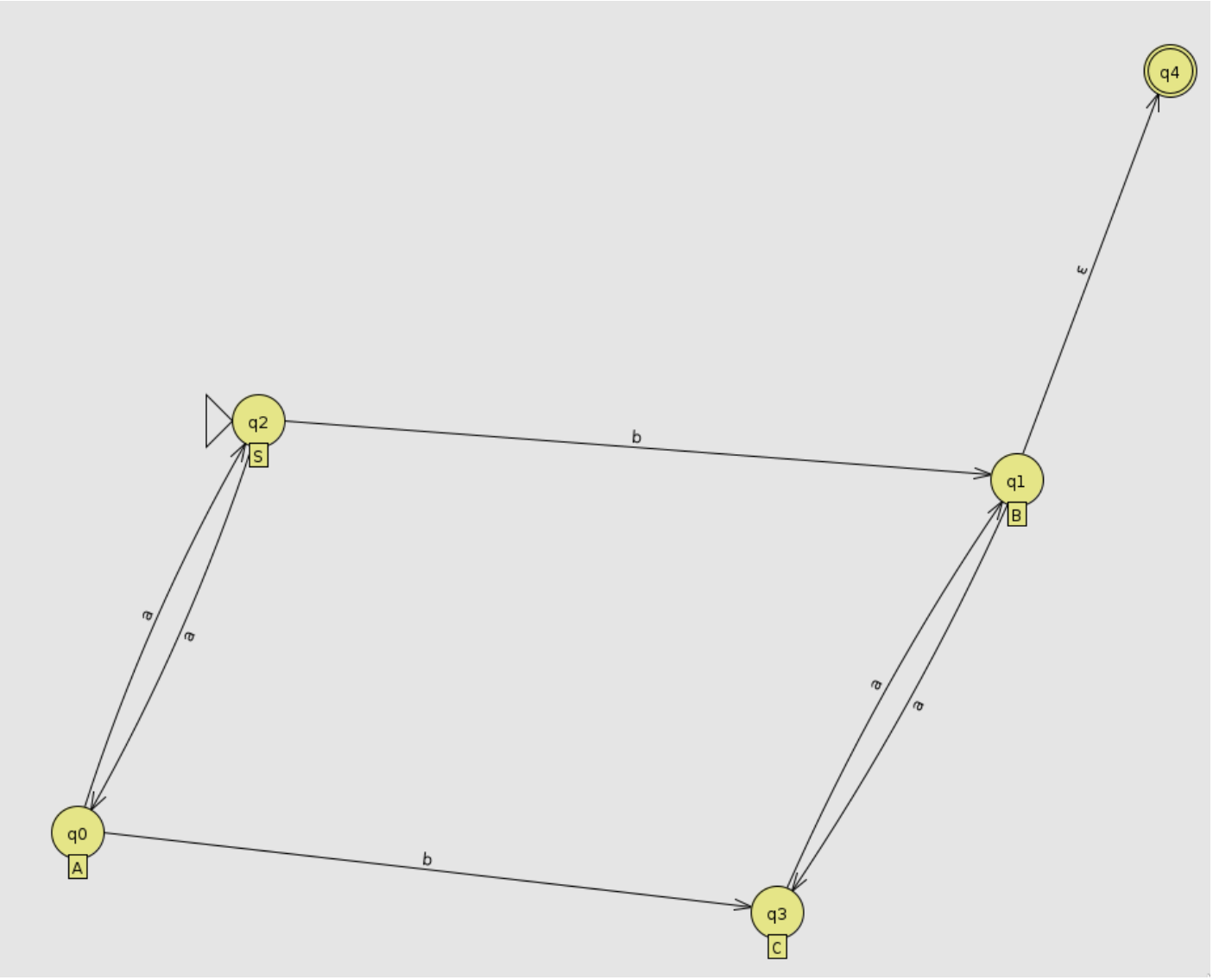
Cet automate de Moore va être défini par le quintuplet $M = (E, \Sigma, t, e_0, A)$ où E est l'ensemble fini d'états, Σ est l'ensemble des inputs (tel que l'alphabet dans le cas des langages formels), t est la fonction de transition d'un état à l'autre, e_0 est l'état initial et A est l'ensemble des états d'acceptations.

Dans le cadre des langages formels, tous les langages reconnus par une machine de Moore seront des grammaires de type 3 (langage régulier). Et on peut créer un automate à partir d'une grammaire ou une grammaire à partir d'un automate.

Comme vu à la page précédente, une grammaire d'un langage peut être définie comme étant $G = (\Sigma, N,P,S)$ où :

- Σ est l'alphabet (c'est-à-dire les inputs)
- N l'ensemble des symboles non-terminaux (c'est-à-dire les états)
- P sont les productions de la grammaire (l'équivalent des transitions entre les états)
- S le symbole non terminal de départ (soit l'état initial)

Voici un automate représentant une grammaire (de type 3, soit régulière) avec la règle suivante "tout mot avec un nombre pair de `a` et strictement 1 `b` "



Ici q4 est un état final, mais cet état aurait très bien pu être q1 (soit B)

Dans cet automate, les états S, A, B, C correspondent respectivement à :

- S = nombre pair de `a` et aucun `b`
- A = nombre impair de `a` et aucun `b`
- B = nombre pair de `a` et 1 seul `b`
- C = nombre impair de `a` et 1 seul `b`

Donc on démarre dans l'état S, car il y a 0 `a` et 0 est un nombre pair, si on ajoute un `a` on passe dans B parce qu'il y a un nombre impair de `a` si on rajoute un `a` on repasse en S parce qu'il y a de nouveau un nombre pair de `a`. Si on ajoute un `b` on passera alors dans l'état B ou C dépendant de l'état d'origine.

Cet automate peut simplement être converti en la grammaire suivante $G = (\{a, b\}, \{ S, A, B, C \}, P, S)$ dont voici les règles de production P :

```
S → aA
S → bB
A → aS
A → bC
B → aC
B → ε
C → aB
```

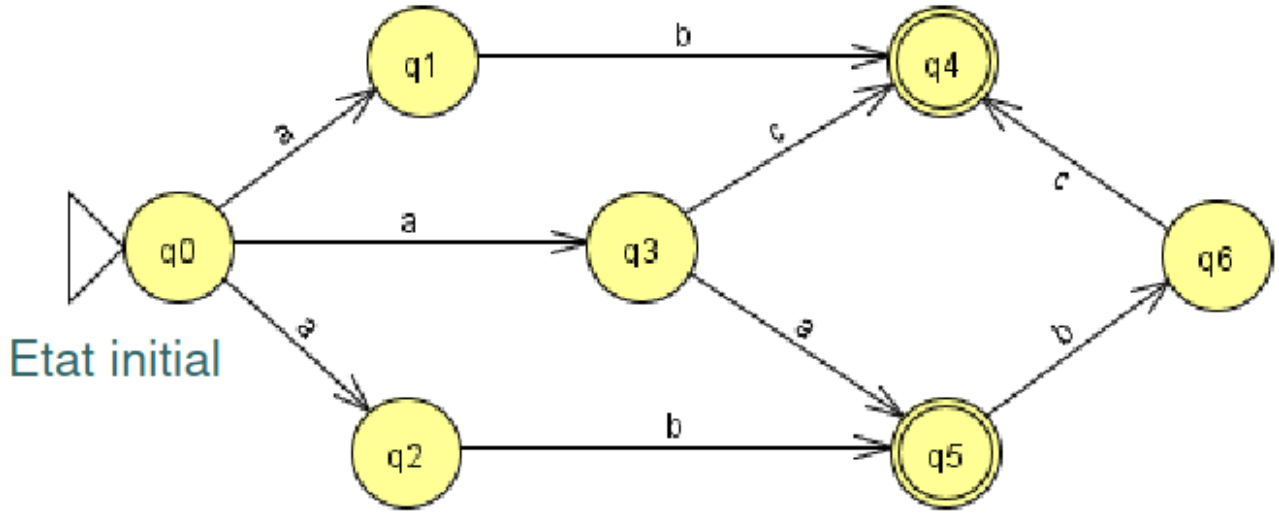
Les automates déterministes et non déterministes

Un automate est dit **déterministe** si depuis un certain input et un certain état actuel, il y a une seule transition possible.

Et il est dit **non déterministe** si depuis un certain input et un certain état actuel, il y a plusieurs transitions possibles ou aucune.

Il est parfaitement possible de faire des automates non déterministes, et il existe un algorithme simple qui permet de convertir un automate non déterministe en automate déterministe.

Pour le convertir, il suffit de faire un tableau de toutes les transitions et de regrouper les états, par exemple en partant de l'automate non déterministe suivant :



> Pro tip: On peut utiliser JFLAP pour convertir un NFA en DFA, pour cela il faut aller dans "Finite automation", créer l'automate puis cliquer sur Convert puis sur "Convert to DFA" puis enfin sur "Complete". On peut aussi tester l'équivalence entre 2 FA (finite automation) en créant un nouveau FA puis en allant dans "Test" puis dans "Test equivalence"

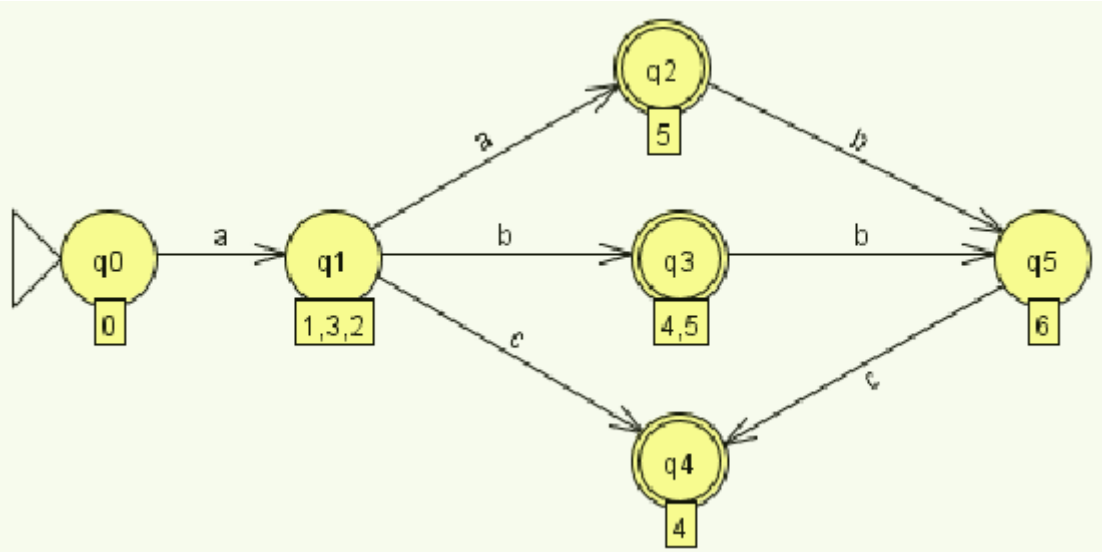
On peut ensuite construire un tableau de toutes les transitions possibles pour chaque input à partir de l'état initial (à noter les états en gras sont les états d'acceptation)

Etats	a	b	c
0 (initial)	1,3,2	∅	∅

Ensuite on peut ajouter pour chaque nouvel état (ou groupe d'états) les transitions correspondantes :

Etats	a	b	c
0 (initial)	1,3,2	∅	∅
1,3,2	5	**4,5**	**4**
5	∅	6	∅
4,5	∅	6	∅
4	∅	∅	∅
6	∅	∅	**4**

On peut ensuite construire le nouvel automate :



Si on veut créer un automate **fini et complet**, on peut alors ajouter un nouvel état "P" (poubelle) dans lequel on va ajouter tous les inputs qui ne mènent à rien (∅ dans le tableau)

À savoir que cet algorithme n'est pas optimisé, il y a des techniques pour l'optimiser, mais on n'en parlera pas ici.

Opérations bit à bit

Les bases de numérations

La base que l'on utilise tout le temps pour compter, c'est la base 10 que l'on appelle décimale (qui comprend les chiffres 0,1, 2,3,4,5,6,7,8,9). Mais il existe d'autres tel que l'hexadécimal (base 16), le binaire (base 2), l'octal (base 8) et bien d'autres.

Base quelconque vers décimal

Pour convertir un nombre d'une base quelconque vers le décimal (base 10) on peut appliquer la formule (que l'on répète pour chaque chiffre)

$$ a_n B^n $$

- n est le numéro de la position du chiffre dans le nombre (unité → 0, dizaines → 1, centaines → 2, dixièmes → -1, centièmes → -2)
- a_n correspond à la valeur du chiffre en position n du chiffre dans sa base d'origine
- B est la base (exemple hexadécimal → 16)

Par exemple, pour convertir le nombre `CAFE` (hexadécimal) en décimal, on peut faire

$$ CAFE_16 = 12 * 16^3 + 10 * 16^2 + 15 * 16^2 + 14 * 16^0 = 51866_10 $$

Décimal vers binaire (ou autre base)

Pour convertir le décimal en binaire, on peut utiliser le resserrement d'intervalle, mais qui plus simplement peut consister à écrire le nombre sur sa calculatrice, puis `EXE`, `÷R`, `2`. Ensuite on peut appuyer sur `EXE` et à chaque fois noter le nombre en "R" (reste) qui correspond aux chiffres binaires **de droite à gauche**. Mais il en va aussi pour convertir vers d'autres bases que le binaire.

Par exemple, si je veux transformer 42 en binaire :

```
Ans = 42        (42 EXE)
Ans ÷R 2        (÷R 2 EXE)
R=0             (EXE)
R=1             (EXE)
```

```
R=0         (EXE)
R=1         (EXE)
R=0         (EXE)
R=1         (EXE)
0
```

Donc si on remet le nombre à l'endroit (donc de haut en bas) ça donne `101010` qui correspond à $2^5+2^3+2^1 = 32+8+2 = 42$

Binaire vers hexadécimal (ou autre base de puissance de 2)

Convertir du binaire en hexadécimal (ou toute autre base en puissance de 2), il suffit d'aligner le binaire et l'hexadécimal et de répartir les bits autrement.

Donc si on veut convertir `101010` en hexadécimal, il faut trouver le nombre de bits (chiffre binaire) auquel un nombre hexadécimal correspond, soit 4, car $\sqrt{16} = 4$. On va donc diviser le nombre en groupe de 4 en partant de la droite (**bit de poids faible** → de plus petite valeur, l'inverse est appelé **bit de poids fort**)

```
10 1010
```

Ensuite on peut compléter à gauche avec des `0`

```
0010 1010
```

Enfin on peut convertir chaque bloc en chiffre hexadécimal :

```
BIN 0010 1010
DEC   2   10
HEX   2   A
```

Donc le nombre hexadécimal final est `2A`, qui correspond exactement à `101010` en binaire, soit `42` en décimal.

Opérations bit à bit (bitwise)

Une opération bit à bit est une opération qui manipule les représentations binaires des données.

Il existe 2 catégories d'opérations, les opérations **logiques** (NOT AND OR et XOR) et les opérations de **décalage** (shift left, shift right)

Les opérations logiques vont être effectuées sur chaque bit, par exemple :

```
a   10101011
b   11001101
    ↓↓↓↓↓↓↓↓ l'opération AND est effectuée sur chaque bit
a&b 10001001
```

Pour voir comment fonctionne les autres connecteurs logiques, voir sur cette synthèse du Q1

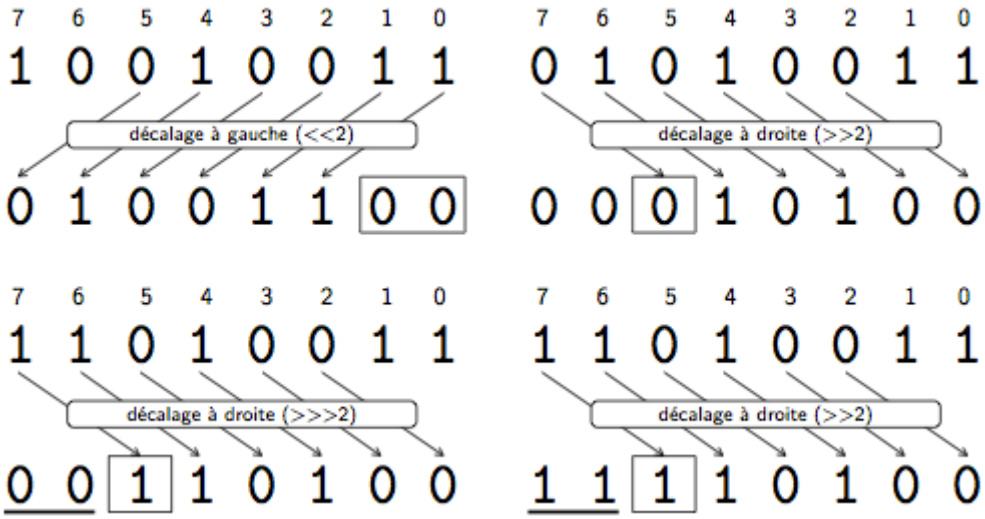
Mais en résumé voici :

a	b	~a (NOT)	a & b (AND)	a | b (OR)	a ^ b (XOR)
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Pour ce qui est des opérations de décalages :

- Dans un **décalage à gauche (<<) de x positions**, on supprime les x premiers bits (si on est sur un nombre de bits limités) et on ajoute x 0 à la fin (bits de poids faible)

- Dans un **décalage à droite signé (ou arithmétique) (>>) de x positions**, on copie le bit de poids fort (le plus à gauche) x fois devant et on supprime les x derniers bits (bits de poids faible)
- Dans un **décalage à droite non signé (>>>) de x positions**, on ajoute x 0 devant et on supprime les x derniers chiffres



Donc si on fait un `>>> 2` on doit faire un décalage à droite non signé 2 fois de suite. Par exemple pour faire `42 >>> 2`, on doit convertir 42 en binaire `101010` puis ajouter deux `0` au début (ce qui donne `00101010`) et supprimer les deux derniers chiffres (ce qui donne `001010`) puis reconvertir en décimal (ce qui donne 10)

```
    42 >>> 2
= 101010 >>> 2
= 001010
= 10
```

Les masques binaires

Les opérateurs sont souvent utilisés avec des "masques" les masques sont des successions binaires destinées à indiquer la position du/des bit(s) concernés par une certaine opération. On peut donc appliquer une opération sur une certaine partie de bits uniquement.

Les 2 masques les plus simples sont les masques qui utilisent des `1` pour signifier les positions particulières telles que `1000` pour spécifier la 3e position (les positions commencent par 0), un tel masque peut être généré avec le décalage `1 << 3`

L'autre type sont les masques inversés qui utilisent des `0` pour signifier les positions telles que `0111` pour spécifier la 3e position, un tel masque peut être généré en inversant le précédent `~(1 << 3)`

Les masques sont généralement désignés par leur valeur hexadécimale, donc `1000` sera appelé `0x8` et `0111` sera appelé `0x9`

Extraire une information

Pour un exemple plus concret, disons que l'on a une couleur RGB. Une couleur RGB sur 32 bits où les octets (soit 8 bits chacun) représentent respectivement rouge, vert et bleu.

```
0x F66FFF
0b 11110110 01101111 11111111
```

Si on veut uniquement avoir le vert (soit les 8 bits du milieu, on peut d'abord utiliser un décalage à droite pour éliminer le bleu (la couleur de droite)

```
11110110 01101111 11111111 >> 8
=      11110110 01101111
```

Ensuite on peut utiliser un masque pour garder seulement les 8 premiers bits de poids faible (donc le vert). On veut donc avoir le masque `11111111` qui peut être représenté comme étant `0xFF`. Enfin, on peut extraire la couleur verte en faisant un opérateur AND entre notre couleur décalée et notre masque

```
11110110 01101111
&      11111111
       ↓↓↓↓↓↓↓↓
=      01101111
```

On sait donc que la couleur verte a une valeur binaire de `01101111`, soit une valeur hexadécimale de `6F` soit une valeur décimale de 111.

Pour faire tout ce qui est décrit plus tôt en Java :

```
int color = 0xF66FFF;
int mask = 0xFF;
int greenValue = (color >> 8) & mask; // donnera 0x6F
//              ^           ^ Ajout du masque pour uniquement avoir les 8 derniers bits (vert)
//              ^ Suppression des 8 derniers bits (bleu)
```

Remplacer une information

On peut aussi utiliser les masque pour remplacer une information à une certaine position. Si on veut remplacer la couleur verte sans changer la couleur rouge ou bleu par exemple. Disons que l'on veut que la valeur de vert soit `0x45`

Pour cela, il va falloir reprendre notre nombre de départ.

```
0x F66FFF
0b 11110110 01101111 11111111
```

Puis on peut appliquer le masque `11111111 00000000 11111111` (`0xFF00FF`) pour faire en sorte que seul la couleur verte soit à 0. En utilisant l'opérateur AND on peut donc mettre le vert à 0.

```
  11110110 01101111 11111111
& 11111111 00000000 11111111
  ↓↓↓↓↓↓↓↓ ↓↓↓↓↓↓↓↓ ↓↓↓↓↓↓↓↓
= 11110110 00000000 11111111
```

Ensuite on peut prendre notre nouveau vert `0x45` et le décaler pour qu'il soit à la même place que le vert dans la couleur de base

```
        01000101 << 8
= 01000101 00000000
```

Enfin on peut combiner les deux avec un OR

```
11110110 00000000 11111111
|       01000101 00000000
        ↓↓↓↓↓↓↓↓ ↓↓↓↓↓↓↓↓
=        01000101 11111111
```

Pour faire la même chose que tout ça en Java :

```
int color = 0xF66FFF; // Couleur de départ
int greenValue = 0x45; // Nouvelle valeur de vert à écrire
int mask = 0xFF00FF; // Masque pour remettre le vert d'origine à 0
int newColor = (color & mask) | (greenValue << 8)
//             ^             ^ ^ Décalage du vert au bon endroit
//             ^             ^ Ajout du vert
//             ^ Suppression du vert dans la couleur d'origine
```

Tester la valeur / égalité de 2 choses

Si on veut savoir si les bits 8 et 9 du nombre de départs valent 1, on peut créer un masque `11 00000000` (`0x300`) et l'appliquer avec.

```
int color = 0xF66FFF;
int mask  = 0x300;
boolean isEqual = (color & mask) == mask
// isEqual is True
```