Architecture et logique

Spécialité numérique et sciences informatiques en classe de première

Lycée Rabelais
Saint Brieuc
2023-2024

Chapitre 1

Turing et Von Neumann

1 Un peu d'histoire

1.1 La machine de Turing

En 1936, Alan Turing publie un article de mathématiques, fruit de ses réflexions sur le thème : « est-il possible de déterminer de manière mécanique si un énoncé mathématique valide est vrai ou non ? » .
C'est une question cruciale : si sa réponse est « oui » cela veut dire qu'il sera peut-être possible de fabriquer une machine qui nous dira si un énoncé (par exemple un théorème qu'on aimerait démontrer) est vrai ou non. Plus besoin de démontrer car la machine le fera à notre place !
Turing est amené à proposer un modèle abstrait de machine de calcul que l'on appelle désormais *machine de Turing*.

Alan Turing (1912-1954) était aussi marathonien.

Il faut donc imaginer une machine qui peut se déplacer sur un ruban aussi étendu qu'on le désire en avançant ou reculant d'une case à la fois.
Définir une machine M c'est se donner

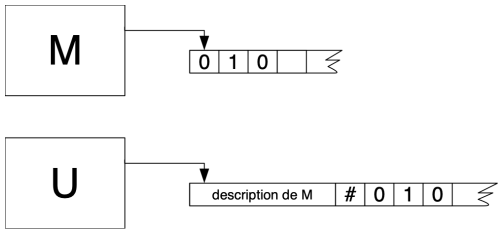
- un ensemble d'*états* $\mathcal{E}$ dans lesquels la machine M pourra se trouver ;
- un ensemble de *symboles* (un alphabet) $\mathcal{A}$, que la machine peut lire ou écrire sur le ruban ;

– un ensemble de *règles* qui décrivent, selon l'état dans lequel M se trouve et le symbole qu'elle lit, quel symbole elle écrit à la place de ce symbole sur le ruban et dans quel sens elle se déplace pour lire le prochain symbole. Cet ensemble de règles peut s'appeler un *programme*.

Exercice 1

Faire l'activité « Machine de Turing » .

La machine décrite précédemment ne possède qu'un seul programme. Turing a donc eu l'idée de ce que l'on appelle maintenant une *Machine de Turing universelle* (MTU).



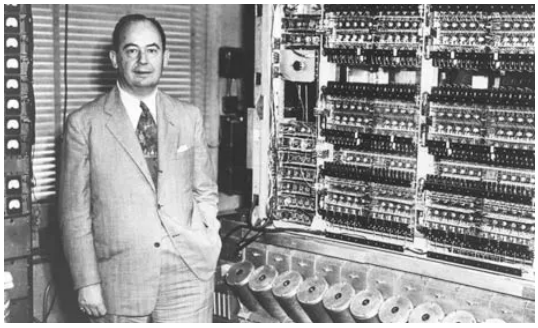
Une machine de Truing universelle.

Il s'agit d'une machine de Turing U qui est capable de simuler n'importe quelle machine de Turing M, pourvu qu'on lui fournisse l'ensemble des règles de M et son état de départ. C'est en quelque sorte l'origine de l'ordinateur programmable, le programme étant les règles de M, et M étant variable.

1.2 L'ordinateur programmable

Au milieu des années 40, John Von Neumann et ses collègues ont mis au point une version concrète de l'ordinateur programmable avec une architecture révolutionnaire pour l'époque, qui reste commune à la plupart des ordinateurs actuels et qui porte le nom d'*architecture de Von Neumann*. Celui-ci a expliqué que l'idée de machine de Turing universelle a directement inspiré le projet.

Parmi les premiers ordinateurs figurent l'ENIAC, ordinateur opérationnel à la fin de l'année 1945 et destiné à effectuer des calculs balistiques. C'est une énorme machine, de 90cm d'épaisseur, 2,40m de haut et ... 30,5m de long, le tout pour un poids de 30 tonnes.



John Von Neumann (1903-1957).

Les premières personnes à programmer cet ordinateur sont six femmes, toutes mathématiciennes. L'ordinateur peur réaliser 100 000 additions par secondes, ou encore 357 multiplications par seconde, ou encore 38 divisions par seconde, le tout avec une capacité mémoire de 20 nombres signés à 10 chiffres en base 10.

Quatre des six programmeuses de l'ENIAC.

On peut voir la taille de quelques-uns des 17 468 tubes à vide qui entraient dans la composition de l'ordinateur. Dès 1947 les transistors ont remplacé les tubes à vides et n'ont cessé d'être miniaturisés. De nos jours les transistors sont directement gravés dans le silicium, leur taille fait quelques nanomètres (10^{-9}m) et une carte graphique RTX 2080 en comporte quasiment 20 milliards. La puissance de calcul a beaucoup augmenté puisque le microprocesseur d'un bon PC actuel a une puissance d'environ 150 GFLOPS, c'est à dire 150 milliards d'opérations sur des nombres en virgule flottante par seconde !

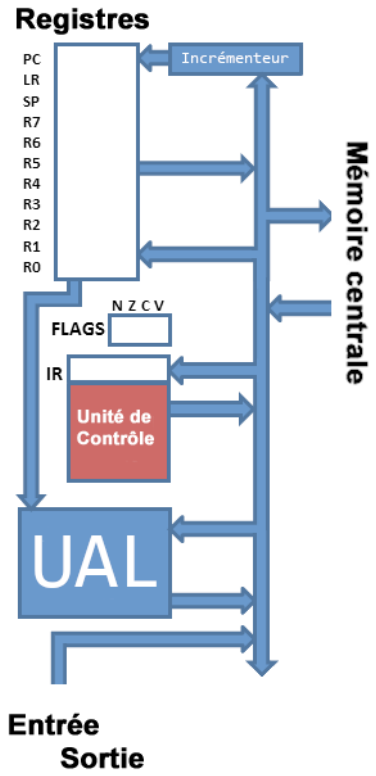
Exercice 2

> Entre un transistor des années 50 (quelques centimètres) et un transistor actuel, quel est le facteur de réduction de taille ?
> Entre l'ENIAC et ses multiplications par seconde et un PC actuel, quel est le facteur d'augmentation de puissance de calcul ?

2 L'architecture de Von Neumann

Elle décompose l'ordinateur en 4 parties distinctes :

- l'*unité arithmétique et logique* ou UAL, dont le rôle est d'effectuer des opérations de base ;
- l'*unité de contrôle* et son registre IR, qui est chargée de séquencer les opérations ;
- la *mémoire* qui stocke à la fois les données à utiliser et le programme que l'unité de contrôle va séquencer.
- les périphériques d'*entrée-sortie* qui permettent de communiquer dans les 2 sens avec l'extérieur.



Modèle simplifié de *microprocesseur* (CPU : *Central Processing Unit*). Les données transitent par des bus (flèches bleues).

Dans le CPU se trouvent des registres :

- PC (*Program Counter*) qui indique à l'unité de contrôle où aller chercher la prochaine instruction ;
- LR (*Link Register*) qui contient l'adresse à laquelle le programme doit revenir dans le cas où on aurait appelé un sous-programme (l'équivalent d'une fonction) ;

– SP (*Stack Pointer*) qui contient l'adresse du sommet de la pile (la pile est un endroit de la mémoire où l'on « empile » et « dépile » des données, comme une pile d'assiettes) ;

Certaines opérations déclenchent des « événements » : quand un résultat est nul, le *flag* (drapeau) Z est mis à un, *et cætera*.
Un processeur donné est capable d'exécuter un nombre d'instructions de base relativement limité. L'ensemble de ces instructions est appelé *langage machine*. Chaque instruction machine est composé d'une ou deux parties :

– un code opération (appelé *opcode*) qui indique le type de traitement à réaliser ;

– les données éventuelles sur lesquelles l'opération doit être réalisée.

Le fonctionnement d'un CPU est cyclique, la fréquence des cycles étant réglée par une horloge (par exemple un processeur moderne cadencé à 3GHz effectue 3 milliards de cycles par seconde).

Déroulement d'un cycle

– le contenu de la RAM pointé par PC est copié dans l'IR de l'unité de contrôle ;

– l'unité de contrôle décode l'instruction qu'on lui donne et la fait exécuter ;

– l'exécution provoque l'utilisation des registres, et/ou une lecture ou écriture dans la RAM, éventuellement un accès aux entrées/sorties.

Les instructions machine étant « désespérément austères » lorsqu'on les écrit en binaire ou en hexadécimal, on les écrit dans un langage compréhensible par les humains. Ce langage s'appelle l'*assembleur*.

Exercice 3

> Faire l'activité : Simulateur de CPU.

Chapitre 2

Logique

1 Du transistor à l'ordinateur

Le transistor est à la base de la plupart des composants d'un ordinateur. Pour faire simple c'est un composant avec une entrée, une sortie, et une alimentation.Quand il est alimenté, le transistor laisse passer le courant de l'entrée vers la sortie et dans le cas contraire le courant ne passe pas.
C'est l'élément de base des *circuits logiques*.

Définition : circuit logique

> Un circuit logique prend en entrée un ou plusieurs signaux électriques. Chacun de ses signaux peut être dans l'état 0 ou l'état 1.
> En sortie, le circuit logique produit un signal (0 ou 1) obtenu en appliquant des *opérations booléennes* aux signaux d'entrée.

Un circuit logique est une implémentation matérielle d'une *fonction logique*. La fonction logique est, quant à elle, la version « mathématique » du circuit. Nous confondrons ces deux notions par la suite.

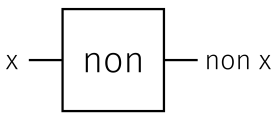
1.1 Opérateurs logiques de base

À l'aide des opérateurs suivants, on peut construire toutes les fonctions logiques.

L'opérateur « non »

C'est un opérateur **unaire** : il ne prend qu'une seule variable booléenne en entrée.

x	*non* x
0	1
1	0



La table de vérité et le symbole de porte européen du **non**.

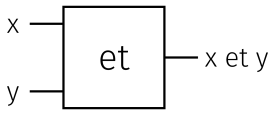
Cet opérateur renvoie « le contraire de ce qu'il a reçu » .
Parmi les notations que l'on rencontre pour noter « non x » il y a

- `NOT x`
- $\overline{x}$
- `!x`

L'opérateur « et »

C'est un opérateur **binaire** : il prend deux variables booléennes en entrée.

x	y	x*et* y
0	0	0
0	1	0
1	0	0
1	1	1



La table de vérité et le symbole de porte européen du `et`.

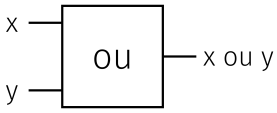
Un « et » ne renvoie vrai que si ses deux entrées sont vraies.
Parmi les notations que l'on rencontre pour noter « x et y » il y a

- `x AND y`
- $x \wedge y$
- `x && y`

L'opérateur « ou »

C'est également un opérateur **binaire**.

x	y	x*ou* y
0	0	0
0	1	1
1	0	1
1	1	1



La table de vérité et le symbole de porte européen du `ou`.

Un « ou » ne renvoie faux que si ses deux entrées sont fausses.
Parmi les notations que l'on rencontre pour noter « x ou y » il y a

- `x OR y`
- $x \vee y$
- `x || y`

On peut montrer qu'il est possible de se passer de la fonction *et* et que toutes les fonctions logiques peuvent s'écrire à l'aide de fonctions *non* et *ou* (on peut même n'utiliser qu'une seule fonction : la fonction « non ou »). Le choix de ces trois fonctions *et*, *ou* et *non* est donc, en quelque sorte, arbitraire.

Définition : Équivalence de deux circuits/fonctions logiques

On dira que deux fonctions logiques sont équivalentes lorsqu'elles prennent le même nombre de variables en entrée (le même nombre de signaux si on parle de circuits) et si ces deux fonctions donnent le même résultat lorsque les variables d'entrées ont les mêmes valeurs : on dit que les fonctions ont même *table de vérité*. Lorsque deux fonctions logiques sont équivalentes, on dit aussi que leurs expressions booléennes sont équivalentes.

Exemples

- Les expressions booléennes not(A and B) et (not A) or (not B) sont équivalentes.
  Pour le prouver il suffit de vérifier que leurs tables de vérité sont les mêmes : on fait varier les valeurs de A et de B selon toutes les possibilités et on regarde le résultat de chaque expression.

A	B	A and B	not (A and B)
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

A	B	not A	not B	(not A) or (not B)
0	0	1	1	1
0	1	1	0	1
1	0	0	1	1
1	1	0	0	0

  Les dernières colonnes de chaque tableau sont les mêmes : les expressions sont donc

équivalentes.

– En procédant de même on montre très facilement que « non non x » et « x » sont équivalentes.

Exercice 4

Montrer que not(A or B) et (not A) and (not B) sont équivalentes.

Exercice 5

On peut définir le « ou exclusif » noté xor comme ceci : A xor B n'est vrai que si A est vrai ou B est vrai, mais pas les deux.

1. Donner la table de vérité de xor.
2. Montrer que A xor B équivaut à (A or B) and (not(A and B)).
3. Montrer que A xor B équivaut à (A and (not B)) or ((not A) and B).
4. Représenter A xor B avec un circuit logique, en utilisant les symboles de porte logique européens.

Exercice 6

On définit l'opération « nor » , notée $\downarrow$ par :

$$A \downarrow B = \textit{not}(A \textit{ or } B)$$

Cette opération est dite *universelle* car elle permet de retrouver toutes les autres opérations.

1. Écrire la table de vérité de nor.
2. Montrer que $A \downarrow A = \textit{not } A$.
3. En utilisant les exemples de la page précédente, en déduire que

$$(A \downarrow B) \downarrow (A \downarrow B) = A \textit{ or } B$$

4. Comment à partir de A, B et $\downarrow$ obtenir A *and* B ?

1.2 Un exemple détaillé : le multiplexeur

Il s'agit d'une fonction très importante : soient A, B et C trois variables logiques, alors m(A,B,C)=B si A vaut 0 et C si A vaut 1.
m permet donc de sélectionner B ou C suivant la valeur de A. Voici la table de valeurs de m :

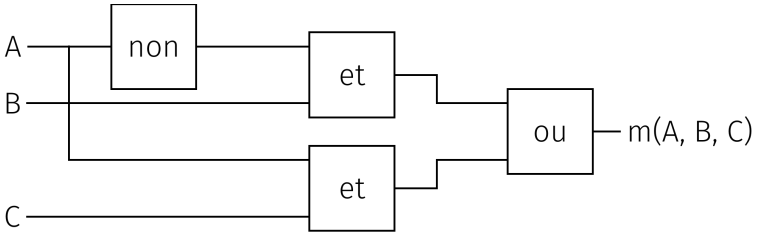
A	B	C	m(A, B, C)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

On va décomposer m à l'aide des opérateurs de base. Ici, un raisonnement simple permet d'y arriver :

- quand A vaut 0, on peut garder la valeur de B en faisant (*non* A) *et* B ;
- quand A vaut 1, on peut garder la valeur de C en faisant A *et* C ;
- il se trouve que ces deux observations vont bien ensemble car quand A vaut 0, A *et* C vaut automatiquement 0, et quand A vaut 1, (*non* A) *et* B vaut automatiquement 0 ;
- on en conclut que l'**expression symbolique** de m est

$$m(A,B,C) = (A \textit{ et } C) \text{ ou } ((\textit{non } A) \textit{ et } B).$$

Le circuit logique modélisant m est le suivant :



Exercice 7

On veut construire un « additionneur » selon le principe suivant :

- A et B représentent deux bits à ajouter

- S et R sont respectivement
  - la somme (sur un bit, donc) de A et B ;
  - la retenue.

Par exemple si A et B valent 1, alors S vaudra 0 et R vaudra 1.

1. Donner les tables de vérités de R et de S.
2. Exprimer R et S en fonction de A et B et des opérations « non » , « ou » et « et » .
3. Représenter R et S avec un (ou des) circuit(s) logique(s), en utilisant les symboles de porte logique européens.

Exercice 8

Pour chiffrer un message, une méthode, dite du masque jetable, consiste à le combiner avec une chaîne de caractères de longueur comparable. Une implémentation possible utilise l'opérateur `XOR` (ou exclusif) dont voici la table de vérité :

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

Dans la suite, les nombres écrits en binaire seront précédés du préfixe `0b`.

1. Pour chiffrer un message, on convertit chacun de ses caractères en binaire (à l'aide du format UNICODE), et on réalise l'opération `XOR` bit à bit avec la clé.

   Après conversion en binaire, et avant que l'opération XOR bit à bit avec la clé n'ait été effectuée, Alice obtient le message suivant :

   ```
   m = 0b 0110 0011 0100 0110
   ```

   a. Le message `m` correspond à deux caractères codés chacun sur 8 bits : déterminer quels sont ces caractères. On fournit pour cela la table ci-dessous qui associe à l'écriture hexadécimale d'un octet le caractère correspondant.

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Exemple de lecture : le caractère correspondant à l'octet codé 4A en hexadécimal est la lettre J.

b. Pour chiffrer le message d'Alice, on réalise l'opération `XOR` bit à bit avec la clé suivante :

```
k = 0b 1110 1110 1111 0000
```

Donner l'écriture binaire du message obtenu.

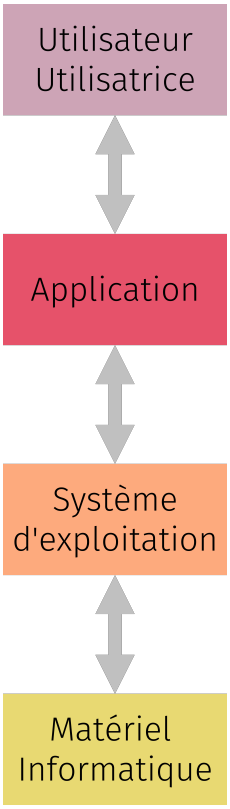
2. a. Donner la table de vérité de l'expression booléenne `(a XOR b) XOR b`.

b. Bob connaît la chaîne de caractères utilisée par Alice pour chiffrer le message. Quelle opération doit-il réaliser pour déchiffrer son message ?

Chapitre 3

Systèmes d'exploitation

1 Qu'est-ce qu'un système d'exploitation ?



Un système d'exploitation (*Operating System* en anglais, abrégé OS) est un ensemble de programmes. Il a plusieurs fonctions :

- c'est un intermédiaire entre l'utilisateur et les applications qu'il utilise (qui sont aussi des programmes) et le matériel ;
- c'est lui qui partage les *ressources* entre les différents programmes en train d'être exécutés voire entre les différents utilisateurs ;
- c'est lui qui protège la machine des applications, et les applications les unes des autres ;
- c'est lui qui gère l'adaptation entre l'*interface* (graphique par exemple) et le matériel.

Les ressources, ce sont entre autres :

- le matériel qui compose l'ordinateur proprement dit (CPU, mémoire, carte graphique…) et les *périphériques* (imprimante, clé usb, clavier) ;
- les fichiers ;
- les éventuelles connexions à des réseaux.

Il existe de multiples OS.

- pour PC : WINDOWS et beaucoup de distributions de LINUX ;
- pour APPLE : MACOS ;
- pour iPhone et iPad : IOS ;
- pour beaucoup de smartphones : ANDROID.

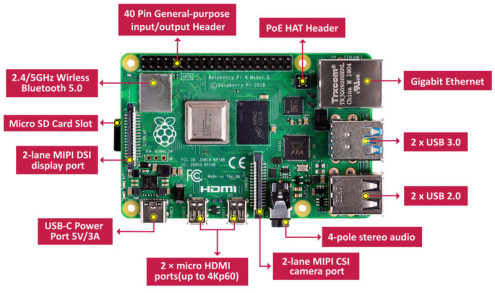
Les appareils connectés ainsi que les « box » des fournisseurs d'accès à Internet et les consoles de jeux disposent aussi de leurs OS. Certains OS sont *gratuits*, d'autres *payants*.

2 Le système de gestion des fichiers

Un *fichier* est un ensemble de données numériques réunies sous un même nom et enregistré sur un support de mémoire permanent appelé *mémoire de masse* (ce peut être un disque dur, un DVD-rom, une clé USB, une carte SD).
Un *système de gestion de fichiers* est une façon de stocker les fichiers sur la mémoire de masse. La plupart des systèmes d'exploitation stockent les fichiers dans des *répertoires* organisés de manière hiérarchique selon une *arborescence* (voir plus bas).

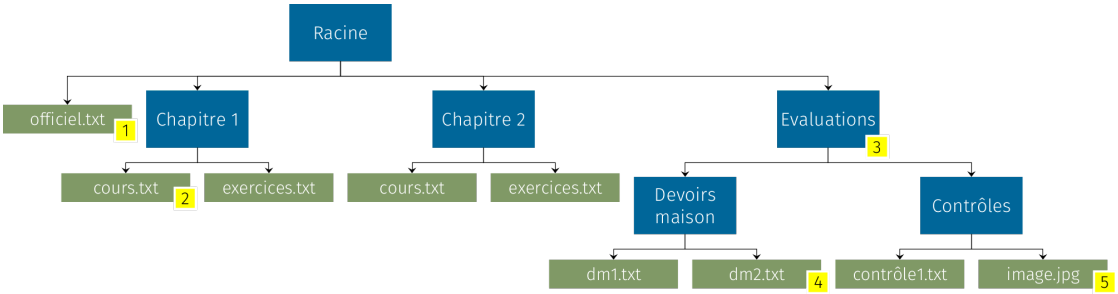
3 L'exemple de Linux et du shell

En 1970 nait UNIX, l'un des premiers système d'exploitation multi-utilisateurs.
Cependant ce système d'exploitation n'est pas *libre*, c'est à dire que son code source n'est pas à la disposition du public.
En 1991, un étudiant finlandais du nom de Linus Torvalds entreprend de créer un clone d'UNIX en le réécrivant totalement.



les Raspberry Pi tournent sous Linux

Ce nouveau système d'exploitation libre s'appelle LINUX et est désormais largement utilisé.
C'est la plupart du temps une version (appelée *distribution*) de LINUX que l'on installe sur les ordinateurs Raspberry Pi.



Exemple d'arborescence de fichiers.

Dans un système de fichier tel que celui de LINUX, la racine se note `/` et les séparateurs (équivalents des flèches du graphique ci-dessus) aussi.
Il y a deux manières d'accéder aux fichiers.

Chemin absolu

On part de la racine et on écrit le chemin pour arriver à l'objet désiré.
Ainsi le chemin absolu de l'objet 1 est `/officiel.txt` et celui de l'objet 5 est `/Evaluations/Contrôles/image.jpg`

Exercice 9

> Donner les chemins absolus des autres objets.

Chemin relatif

On part d'un emplacement précis (à l'intérieur d'un répertoire) et l'on indique le chemin pour aller à l'objet voulu. S'il faut remonter d'un cran dans l'arborescence on utilise na notation `..` comme ceci :

- si l'on est dans le répertoire `Evaluations` et qu'on veut accéder à `dm2.txt` alors son chemin relatif est `Devoirs maison/dm2.txt`;

- si l'on est dans `Contrôles` et qu'on veut accéder au exercices du chapitre 1, le chemin relatif est `../../Chapitre 1/exercices.txt` car il faut d'abord remonter l'arborescence de 2 crans.

Exercice 10

> On est dans le répertoire `Contrôles`, donner les chemins relatifs des objets 1 à 5.

3.1 Le shell

Nous sommes habitués à un environnement graphique pour gérer les copies et déplacements de nos dossiers (et bien d'autre choses comme par exemple sélectionner des fichiers à compresser) mais ce n'est pas la seule manière. Dans tout système d'exploitation muni d'un environnement graphique, on peut trouver un *terminal* (aussi appelé *invite de commande*, ou *shell*) parce que

- c'est avec cette interface textuelle minimaliste que l'on interagissait avec l'OS avant ;
- finalement quand on sait (très bien) s'en servir, on peut faire beaucoup plus de choses plus rapidement avec un shell et au clavier qu'avec un clavier, une souris et un environnement graphique.

Exercice 11

Faire l'activité *utiliser le shell*.

Chapitre 4

Unité et diversité des langages

1 Des éléments communs

Un langage de programmation sert à traduire des algorithmes pour les exécuter sur un ordinateur. Il est composé

- d'un alphabet (ensemble de lettres et de symboles) ;
- d'un vocabulaire (les *mots-clés* du langage) ;
- d'une grammaire (la *syntaxe* du langage).

Les notions suivantes sont communes à l'immense majorité des langages :

- une *instruction* est un ordre donné ;
- une *variable* est un nom qui fait référence à une donnée manipulée par le programme et susceptible de changer au cours de celui-ci ;
- une *constante* est un nom qui fait référence à une valeur immuable ;
- un *type* qui sert à classifier une variable ou une constante et conditionne les opérations qu'il est possible d'effectuer ;
- la *déclaration* consiste à renseigner le traducteur du programme sur la nature des données du programme (type, valeur).
- les *structures de contrôle* telles que le *test* et les *boucles*.
- une *fonction* (ou procédure, ou méthode) sert à isoler un fragment de programme pour pouvoir l'utiliser (éventuellement plusieurs fois) de manière paramétrée.

2 Des différences

2.1 Différences formelles

Tous les langages n'utilisent pas la même syntaxe. En examinant un même algorithme écrit dans plusieurs langages, on constate que

- L'affectation peut être signifiée par `=` (comme en PYTHON, BASIC, C, FORTRAN…) ou par `:=` (ADA, ALGOL, GO pour partie…). On utilise aussi `<-` en CAML.
- Les *listes* (ou tableaux) sont très souvent indicées à partir de zéro… Mais pas toujours (FORTRAN, LUA).
- Très souvent, les structures de contrôles sont délimitées par des accolades (C, JAVA, KOTLIN…) ou par des `begin` et des `end` (RUBY, PASCAL…). Parfois on utilise des variantes du `end` telles que `done`, `END-IF` ou `NEXT` pour signifier une fin de boucle « pour » .

Certains langages ont des syntaxes très similaires : le langage C, JAVA et JAVASCRIPT par exemple (notamment le fait qu'un point-virgule termine une ligne).
D'autres ont une syntaxe et une mise en forme particulière, éloignée de tous les autres, comme le BASIC ou le COBOL.

2.2 Différences structurelles

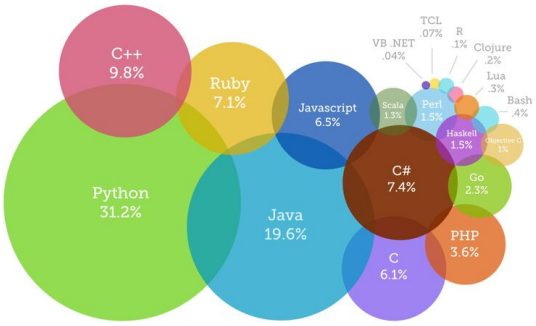
Certains langages obligent à déclarer le type des variables lors de leur création. C'est le cas de C ou JAVA. D'autres obligent même à renseigner les variables et leur type avant toute chose, comme ADA, ALGOL, COBOL. Ce n'est pas le cas en PYTHON ou en RUBY.

Une autre différence majeure vient de la manière dont est traité le programme par le traducteur :

- En C ou en C++, le programme est transformé en *langage-machine* par un *compilateur*. L'intérêt est que l'on gagne en rapidité lors de l'exécution du programme.
- En PYTHON le programme est *interprété* lors de son exécution, ligne par ligne.
- Beaucoup de langages utilisent un procédé hybride : le langage utilise une machine virtuelle, et les programmes sont compilés dans le langage de cette machine virtuelle, qui sera ensuite exécuté. La compilation peut être faite avant et le résultat stocké dans un fichier, ou bien être faite à la volée (*Just In Time compilation*).

Chaque langage suit un ou plusieurs *paradigmes* de programmation qui change radicalement la manière d'écrire les programmes.
Chaque langage suit un ou plusieurs *paradigmes* de programmation qui change radicalement la manière d'écrire les programmes.
Nous avons vu la programmation *impérative* (séquentielle, linéaire) où le programme effectue une liste d'instructions pas-à-pas. Il existe la programmation *évènementielle* lors de laquelle on précise à PROCESSING ce qu'il doit faire quand tel ou tel évènement se produit.
D'autres paradigmes de programmation telle la programmation *orientée objet* ou la programmation *fonctionnelle* sont très utilisés.
Enfin on distinguera des différences dans les contexte d'utilisation de chaque langages : il existe des langages généralistes, des langages qui sont censés être exécutés sur des « serveurs de

pages web » tels PHP, d'autres qui (à la base) ont été conçus pour être exécutés au sein même d'un navigateur (JAVASCRIPT).

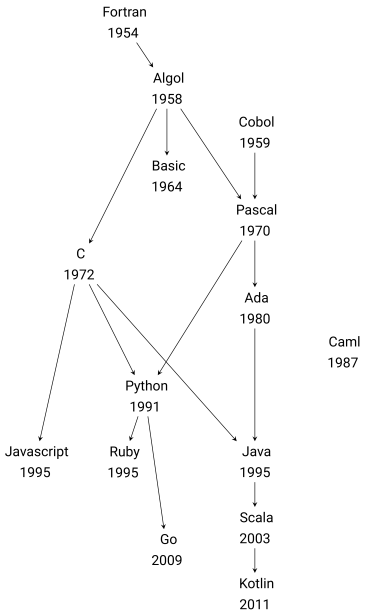


Langages les plus appris en 2019.
Source : `learnworthy.net`.

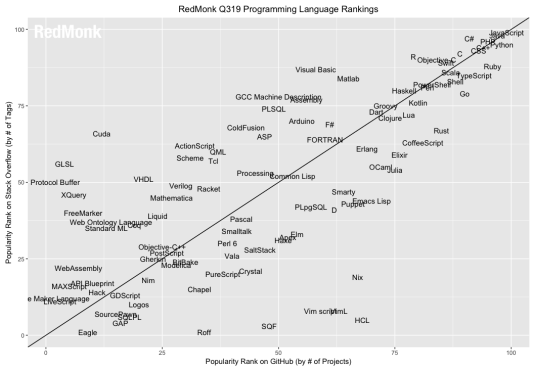
3 Évolution au cours du temps

Les progrès réalisés sur les analyseurs de code permettent de créer des langages avec une syntaxe de plus en plus épurée courte et pour lesquels il n'est pas obligé de déclarer le type de chaque variable.
Les nouveaux langages s'inspirent souvent de leurs prédécesseurs.

On continue d'inventer de nouveaux langages : ceux-ci sont crées en fonction des besoins de l'époque.
De nos jours, les langages qui permettent de programmer facilement plusieurs tâches simultanées (comme par exemple récupérer des données sur INTERNET et en même temps traiter ces données) ont le vent en poupe.



Voici un graphe indiquant comment les langages anciens ont influencé les nouveaux.



Stack Overflow est un forum d'entraide à la programmation.
GitHub est un service d'hébergement de projets logiciels.
Le graphique suivant indique la popularité de différents langages sur ces deux sites.
Source : `redmonk.com`.