Introduction à l'algorithmique et la complexité (et un peu de CAML)

Conception d'algorithmes efficaces (rapides)

Nicolas Nisse

Université Côte d'Azur, Inria, CNRS, I3S, France

Cours dispensés en MPSI (option Info) au CIV, depuis 2011-

http://www-sop.inria.fr/members/Nicolas.Nisse/lectures/prepa/

Outline

Qu'est-ce qu'un Algorithme ?

Definition: Algorithme

Séquence d'opérations (élémentaires) qui, étant données des entrées, calcule une solution/sortie valide.

Point clé 1: la séquence d'opérations est **non ambigüe** / **systématique** (l'ordre des opérations, la définition de chaque opération... sont parfaitement définis)

Exemple: recette de cuisine

Point clé 2: si l'entrée est du bon **type**, la sortie doit être valide (celle attendue).

L'algorithme est alors correct.

Ex.: si vous avez des œufs (quels qu'ils soient), du lait (en bonne quantité)... vous voulez une île flottante, pas une bouillabaisse... Au contraire, si vous n'avez que du poisson, n'espérez pas une île flottante...

Exemples d'algorithmes "de tous les jours"

Démarrer une voiture

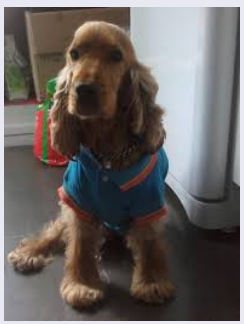
1) prendre la clé ; 2) ouvrir la voiture ; 3) s'asseoir et régler siège et rétroviseurs ; 4) insérer et tourner la clé ; 5) baisser le frein à main ; 6) bidouiller les pédales...

si **l'ordre** est modifié, ça peut mal se passer...

si l'entrée n'est pas du **bon type** (e.g., un vélo au lieu d'une voiture), vous n'obtiendrez pas le résultat espéré...

Vous, au réveil...

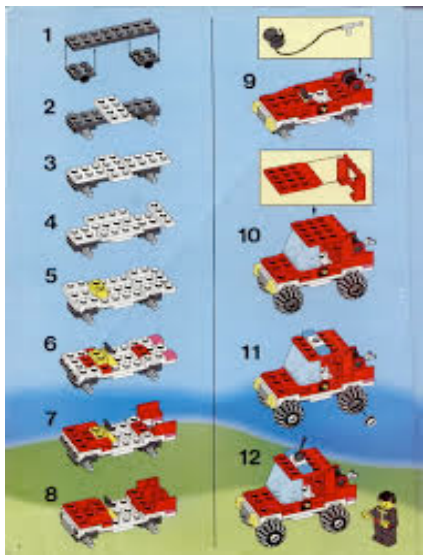
1) le réveil sonne ; 2) attendre 5 minutes en ralant ; 3) petit-déjeuner ; 4) prendre sa douche ; 5) enfiler un tee-shirt ; 6) enfiler un pull...



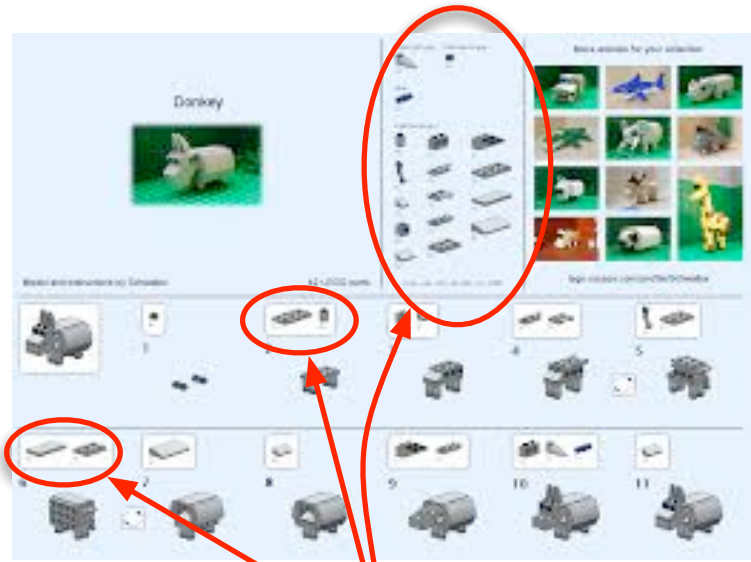
si **l'ordre** est modifié ou si l'entrée n'est pas du **bon type** (pas vous), ça peut mal se passer...

Exemples d'algorithmes "de tous les jours" (2/2)

Notices de montage...



Les notices de montage LEGO/IKEA... sont autant d'exemples de séquences d'instructions pour obtenir un résultat désiré.
Elles peuvent cependant être **Ambigües** (basées sur des images) et seraient **difficilement interprétées par un ordinateur.**



Depuis ``peu", les notices LEGO précisent les ``entrées" de chaque instruction.

J'insiste : les types des entrées, l'ordre et la définition des opérations sont FONDAMENTAUX pour que votre algorithme soit CORRECT

Algorithme : Spécification et Instructions

Un algorithme décrit un procédé, susceptible d'une réalisation mécanique, pour résoudre un problème donné. Il consiste en une spécification (ce qu'il doit faire) et une méthode (comment il le fait) :

- La spécification précise les données d'entrée avec les **préconditions** que l'on exige d'elles (entre autres, le type de données en entrées), ainsi que les données de sortie avec les **postconditions** que l'algorithme doit assurer (entre autres, le type de sortie). Autrement dit, les préconditions définissent les données auxquelles l'algorithme s'applique, alors que les postconditions définissent le résultat auquel il doit aboutir.
- La **méthode** consiste en une suite finie d'instructions, dont chacune est soit une instruction primitive (directement exécutable sans explications plus détaillées) soit une instruction complexe (qui se réalise en faisant appel à un algorithme déjà défini). En particulier chaque instruction doit être **exécutable de manière univoque/systématique**, et ne doit pas laisser place à l'interprétation ou à l'intuition.

Algorithmique *versus* Programmation

L'**algorithmique** consiste à définir et organiser des opérations pour réaliser une tâche. De préférence, on voudra concevoir un algorithme *efficace* (rapide, utilisant peu de mémoire...).

La **programmation** sert à *traduire* un algorithme en un langage (C, C^{++}, Java, Python, CAML, LISP...) que "comprend" un ordinateur.

Parfois (souvent ?), l'efficacité en pratique d'un algorithme dépend du langage de programmation.

Dans la suite, nous illustrons nos algorithmes par une traduction en CAML. Ce qui suit ne prétend pas être un cours de CAML, mais une aide pour pouvoir s'en servir. Nous en exposerons les bases (parfois informellement) en insistant sur les erreurs classiques faîtes par les élèves dans les DS.

Outline

Bases algorithmiques I : Types et Opérations élémentaires

Ex : En "cuisine", les types de données (d'aliments) sont "clairs": e.g., œufs, jambon, cuisse de poulet, haricots verts.... De même, pour chaque type, des opérations possibles sont bien définies (casser un œuf, monter un blanc d'œuf en neige, écosser des petits pois...).

Les principaux **types** de données et opérations élémentaires correspondantes sont :

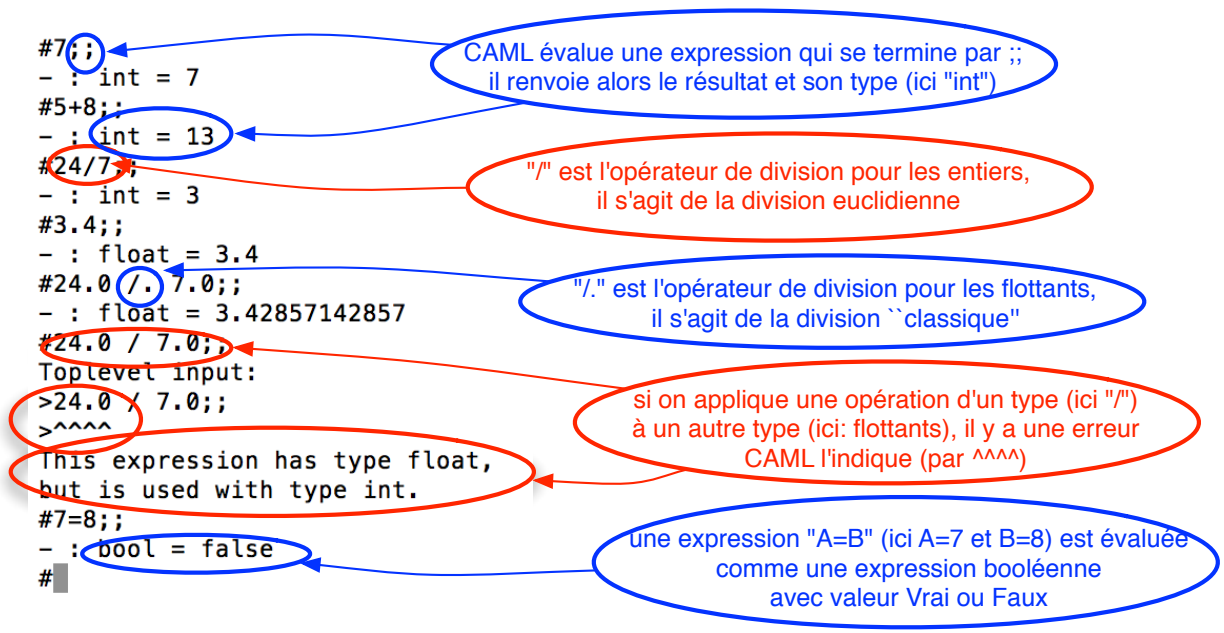
Types (basiques)	**Opérations élémentaires**
booléen (**bool**) $\in \{True, False\}$	opérations bouléennes $=, \vee, \wedge, \neg$
entier (**int**) $\in \{0; 1; 2; 3; \cdots\} = \mathbb{N}$	opérations arithmétiques $+; -; *; /$ comparaisons $(\leq, \geq, <, >, =)$ de 2 entiers
flottant ("réel", **float**) : *ex:* 1.002; ...	$+.; -.; *.;$... comparaisons...
chaîne de caractères (**string**) : *ex:* "*algo*" ,"*AAAB*" ,"*aab*" ...	concaténation, longueur de la chaîne
liste (**list**) (séquence d'éléments généralement d'un même type)	créer liste vide ; ajouter un élément en tête de liste ; prendre la tête de liste ; prendre la queue de liste ; concaténer 2 listes, déterminer longueur de la liste...
tableau (**vect**) (séquence, **de taille fixée**, d'éléments généralement d'un même type)	créer un tableau de longueur ℓ ; affecter une valeur à une "case" d'indice donné ; déterminer la valeur d"une "case" d'indice donné ; déterminer la longueur du tableau

Bases algorithmiques I : Types et Opérations élémentaires première prise de contact avec CAML (int, float, bool)

CAML est un langage **interprété** (par opposition aux langages **compilés**), i.e., l'ordinateur "traduit au fur et à mesure" les instructions.

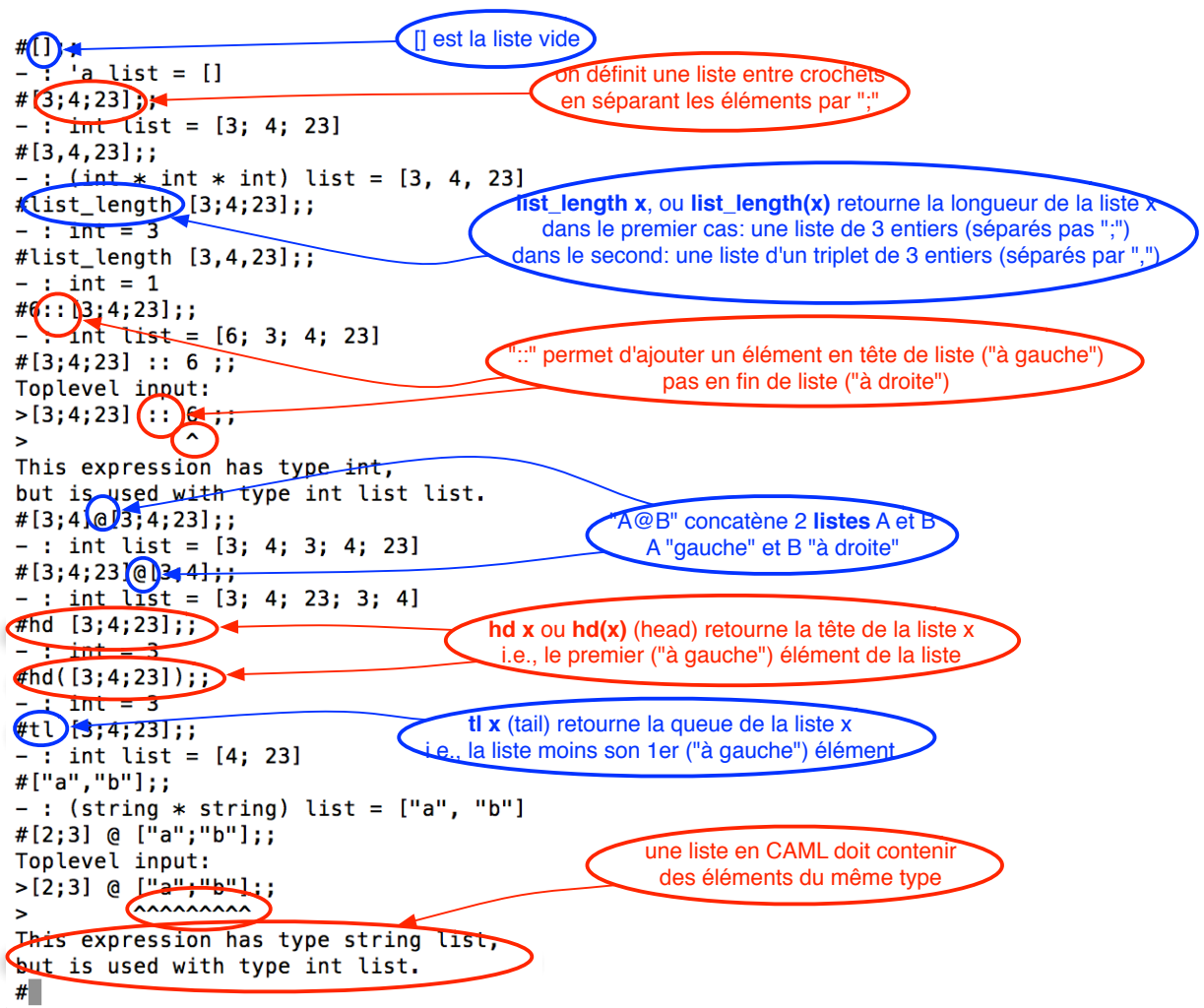
Le début de ligne est indiqué par #. On écrit son instruction (e.g., $5+8$) puis ";;", et en tapant "ENTER", CAML évalue l'expression et renvoie le résultat.

```
>        Caml Light version 0.80

#7;;
- : int = 7
#5+8;;
- : int = 13
#24/7;;
- : int = 3
#3.4;;
- : float = 3.4
#24.0 /. 7.0;;
- : float = 3.42857142857
#24.0 / 7.0;;
Toplevel input:
>24.0 / 7.0;;
>^^^^
This expression has type float,
but is used with type int.
#7=8;;
- : bool = false
#not(((7=8)&&(3=3))or(4=3));;
- : bool = true
#
```



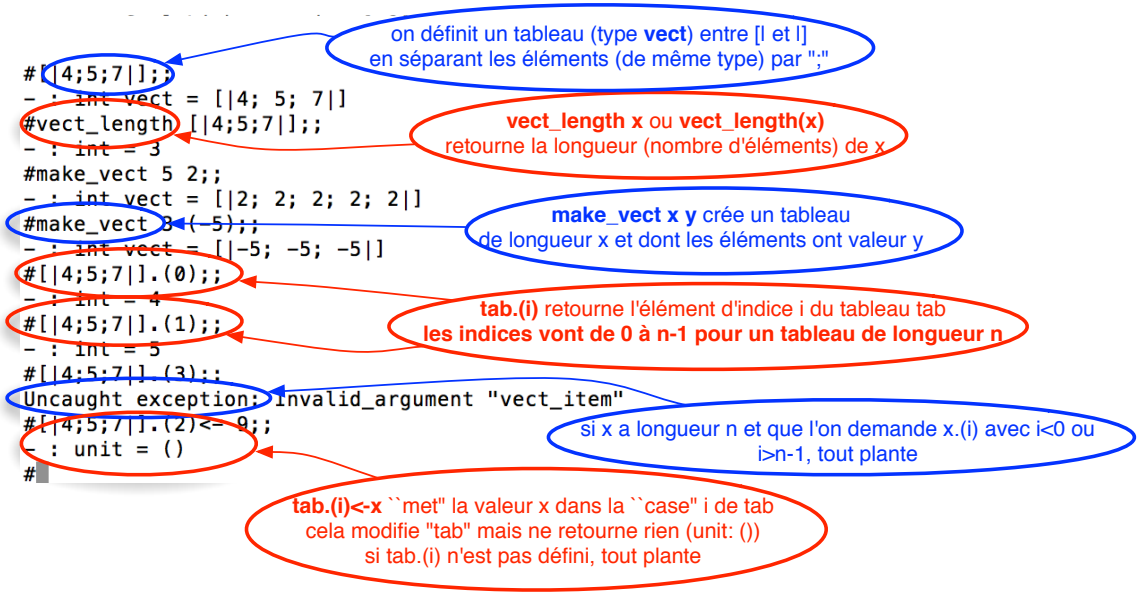
Bases algorithmiques I : Types et Opérations élémentaires première prise de contact avec CAML (list)

```
>       Caml Light version 0.80

#[];;
- : 'a list = []
#[3;4;23];;
- : int list = [3; 4; 23]
#[3,4,23];;
- : (int * int * int) list = [3, 4, 23]
#list_length [3;4;23];;
- : int = 3
#list_length [3,4,23];;
- : int = 1
#6::[3;4;23];;
- : int list = [6; 3; 4; 23]
#[3;4;23] :: 6 ;;
Toplevel input:
>[3;4;23] :: 6 ;;
>            ^
This expression has type int,
but is used with type int list list.
#[3;4]@[3;4;23];;
- : int list = [3; 4; 3; 4; 23]
#[3;4;23]@[3;4];;
- : int list = [3; 4; 23; 3; 4]
#hd [3;4;23];;
- : int = 3
#hd([3;4;23]);;
- : int = 3
#tl [3;4;23];;
- : int list = [4; 23]
#["a","b"];;
- : (string * string) list = ["a", "b"]
#[2;3] @ ["a";"b"];;
Toplevel input:
>[2;3] @ ["a";"b"];;
>        ^^^^^^^^^
This expression has type string list,
but is used with type int list.
#
```



Bases algorithmiques I : Types et Opérations élémentaires première prise de contact avec CAML (vect)

```
>       Caml Light version 0.80

#[|4;5;7|];;
- : int vect = [|4; 5; 7|]
#vect_length [|4;5;7|];;
- : int = 3
#make_vect 5 2;;
- : int vect = [|2; 2; 2; 2; 2|]
#make_vect 3 (-5);;
- : int vect = [|-5; -5; -5|]
#[|4;5;7|].(0);;
- : int = 4
#[|4;5;7|].(1);;
- : int = 5
#[|4;5;7|].(3);;
Uncaught exception: Invalid_argument "vect_item"
#[|4;5;7|].(2)<- 9;;
- : unit = ()
#
```



- **Les différences fondamentales entre liste et tableau** sont qu'on ne peut pas modifier la taille d'un tableau (alors qu'on peut ajouter un élément à une liste ou concaténer 2 listes pour obtenir une liste plus grande). A l'inverse, on ne peut accéder directement à n'importe quel élément d'une liste (sauf la tête) alors que c'est possible pour un tableau.

$\Rightarrow$ Préférer un tableau ou une liste dépend de l'application.

Bases algorithmiques II : Variables

Le problème dans les exemples ci-dessus est qu'on ne stocke pas les résultats des instructions, on ne peut donc pas les utiliser pour les instructions suivantes.

Pour pallier cela, on introduit la notion de variables.

Ex : En "cuisine", les ingrédients ne sont pas suffisants pour réaliser une recette. Il faut également des ustensiles pour stocker les résultats intermédiaires : un saladier pour les blancs en neige, une casserole pour le fond de veau...
De même qu'on prépare ses ustensiles avant de cuisiner, en programmation :

On **déclare** (définit) les variables avant de les utiliser.

En CAML...

Let nom_variable = expression ; ;

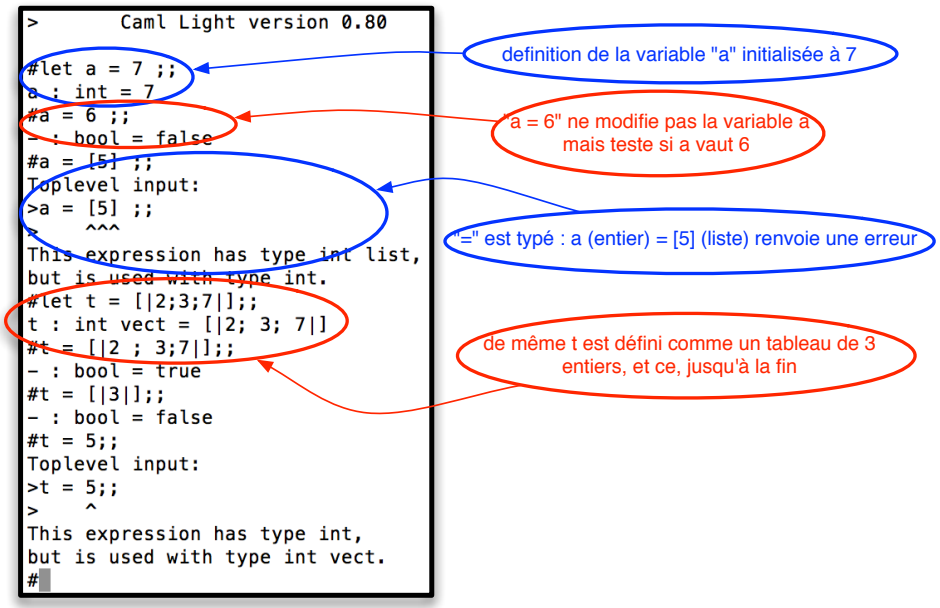
```
>        Caml Light version 0.80

#let a = 5 ;;
a : int = 5
#a ;;
- : int = 5
#let l = [6;3;9];;
l : int list = [6; 3; 9]
#l ;;
- : int list = [6; 3; 9]
#hd l ;;
- : int = 6
#l ;;
- : int list = [6; 3; 9]
#tl l ;;
- : int list = [3; 9]
#
```

definition de la variable "a" ("l") initialisée à l'entier 5 (à la liste [6;3;9])

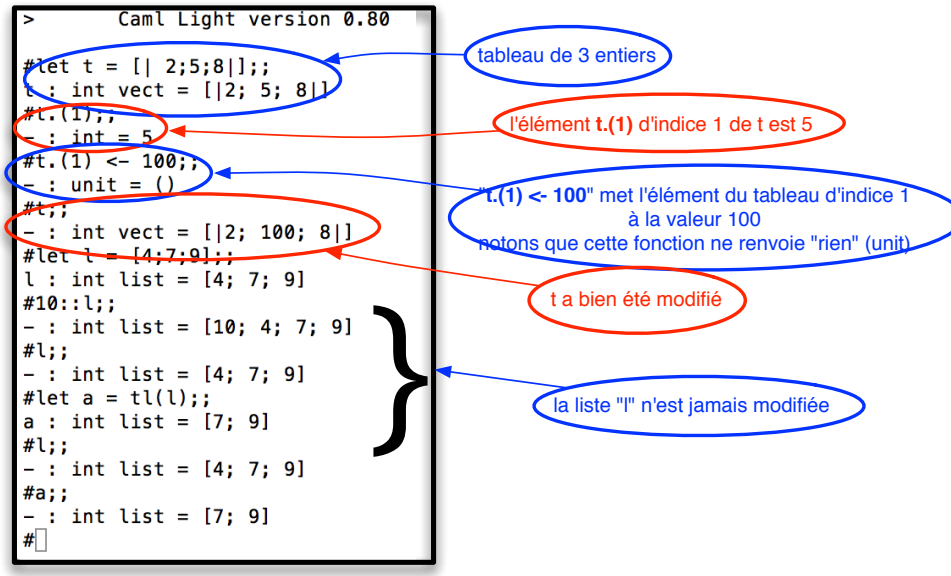
on peut donc s'en servir : par exemple prendre la tête de "l"

Bases algorithmiques II : Variables
première prise de contact avec CAML

En CAML, les variables sont en général **non mutables** (on ne peut pas les modifier).



Le contenu d'un tableau est **mutable**.
Les listes ne le sont pas



Bases algorithmiques II : Variables
première prise de contact avec CAML

Pour pouvoir utiliser des variables modifiables, on utilise des **références**.
Grosso modo, on définit une variable (**pointeur**) qui représente l'adresse d'une case mémoire contenant ce qui nous intéresse.
Il est alors possible de modifier le contenu de ce vers quoi pointe la variable.

```
>        Caml Light version 0.80

#let a = 5 ;;
a : int = 5
#let b = ref 5 ;;
b : int ref = ref 5
#a ;;
- : int = 5
#b ;;
- : int ref = ref 5
#!b ;;
- : int = 5
#b := 7 ;;
- : unit = ()
#b;;
- : int ref = ref 7
#!b;;
- : int = 7
#a := 7;;
Toplevel input:
>a := 7;;
>^
This expression has type int,
but is used with type 'a ref.
#
```

"a" est un entier

"b" est une **référence** sur un entier

"!b" renvoie le CONTENU de la variable pointée par b
le POINT d'EXCLAMATION est important

Il est possible de modifier le CONTENU de la variable POINTEE par une référence avec :=

on peut modifier le contenu d'une variable en utilisant son contenu courrant

```
>        Caml Light version 0.80

#let i = ref 10 ;;
i : int ref = ref 10
#i := !i +3 ;;
- : unit = ()
#!i ;;
- : int = 13
#let l = ref [7;9;2];;
l : int list ref = ref [7; 9; 2]
#l := tl(!l) ;;
- : unit = ()
#!l;;
- : int list = [9; 2]
#i := hd(!l) ;;
- : unit = ()
#!i ;;
- : int = 9
#
```

Bases algorithmiques III : Fonctions
première prise de contact avec CAML

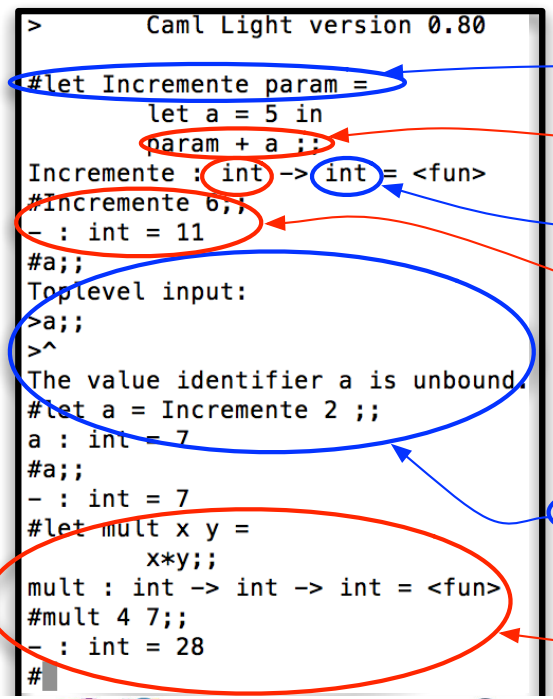
Maintenant que vous connaissez les "briques de base" de CAML, il est possible de les assembler pour créer des **fonctions**.

En CAML...

Let nom_function Paramètre1 Paramètre2 ... =
algorithme ; ;

Le/les paramètre(s) d'entrée peuvent être utilisé(s) dans la fonction.

```
>       Caml Light version 0.80

#let Incremente param =
        let a = 5 in
        param + a ;;
Incremente : int -> int = <fun>
#Incremente 6;;
- : int = 11
#a;;
Toplevel input:
>a;;
>^
The value identifier a is unbound.
#let a = Incremente 2 ;;
a : int = 7
#a;;
- : int = 7
#let mult x y =
        x*y;;
mult : int -> int -> int = <fun>
#mult 4 7;;
- : int = 28
#
```



fonction "Incremente" qui prend en entrée un paramètre "param"

"param" est implicitement un entier puisqu'il est additionné avec un entier

la fonction renvoie l'entier "param+a"="param"+5

application de la fonction "Incrément" à l'entrée (paramètre) 6

Une variable a UNE DUREE DE VIE ici, "a" définie dans la fonction n'est plus utilisable en dehors de la fonction

fonction qui prend deux entrées (x et y) et retourne leur produit

Bases algorithmiques III : Fonctions première prise de contact avec CAML

Une fonction peut en appeler une autre (si elle est déjà définie)

```
>        Caml Light version 0.80

#let TroisFoisPlusUn i =
        let j = i*3 in
        PlusUN j ;;
Toplevel input:
>        PlusUN j ;;
>        ^^^^^^
The value identifier PlusUN is unbound.
#let PlusUN i =
        i +1 ;;
PlusUN : int -> int = <fun>
#let TroisFoisPlusUn i =
        let j = i*3 in
        PlusUN j ;;
TroisFoisPlusUn : int -> int = <fun>
#TroisFoisPlusUn 8;;
- : int = 25
#
```

le mot clé "in" permet de définir des variables/functions dans une function

la fonction PlusUN n'est pas définie

la fonction PlusUN étant définie
la fonction TroisFoisPlusUn prend
un entier i en entrée et renvoie 3i+1

Le/les paramètre(s) d'entrée peuvent être modifié(s)

Une fonction très utile: Echange des éléments d'indice i et j dans un tabelau t

En cuisine : Vous avez deux saladiers, un petit contenant de la salade et un grand avec des tomates. Vous voulez mettre la salade dans le grand saladier. Il va vous falloir un récipient intermédiaire (une variable) pour faire le transvasement. C'est pareil ici !

```
>        Caml Light version 0.80

#let Echange t i j =
        let transvasement = t.(i) in
        t.(i) <- t.(j) ;
        t.(j) <- transvasement ;;
Echange : 'a vect -> int -> int -> unit = <fun>
#let t = [|3 ; 5; 6|];;
t : int vect = [|3; 5; 6|]
#Echange t 0 2 ;;
- : unit = ()
#t;;
- : int vect = [|6; 5; 3|]
#
```

la fonction Echange ne renvoie rien (uni
mais modifie son entrée "t"

Bases algorithmiques III : Fonctions première prise de contact avec CAML

Court (et informel) aparté technique pour rappeler qu'**en CAML, tout est typé**.

```
>       Caml Light version 0.80

#let PlusUN x = x+1;;
PlusUN : int -> int = <fun>
#PlusUN 4;;
- : int = 5
#PlusUN(4);;
- : int = 5
#let Addition x y = x+y;;
Addition : int -> int -> int = <fun>
#Addition 4 5;;
- : int = 9
#Addition(4,5);;
Toplevel input:
>Addition(4,5);;
>         ^^^
This expression has type int * int,
but is used with type int.
#let Addition2 (x,y) = x+y;;
Addition2 : int * int -> int = <fun>
#Addition2 (4,5);;
- : int = 9
#Addition2 4 5;;
Toplevel input:
>Addition2 4 5;;
>          ^
This expression has type int,
but is used with type int * int.
#let PlusX y = Addition 5 y;;
PlusX : int -> int = <fun>
#PlusX 7;;
- : int = 12
#PlusX(7);;
- : int = 12
#
```

fonction qui associe un entier à un entier x

fonction qui, à un entier x, associe une fonction (dépendant de x) qui associe un entier à un entier y

fonction qui associe un entier à un couple d'entiers (x,y)

Notez la différence entre Addition et Addition2 (en particulier, dans la manière de les utiliser)

Outline

Bases algorithmiques IV : Boucles (pour les nuls)

Très (très très...) informellement, les boucles sont des outils indispensables permettant de "factoriser" l'écriture des algorithmes ou de "généraliser" la fonction à réaliser.

Boucle conditionnelle **If** : La plus simple et naturelle des boucles : prenez 2 recettes qui ne diffèrent que par un ingrédient, pas besoin d'écrire deux recettes : SI vous préférez les cerises, ALORS mettez des cerises dans votre clafoutis, SINON, vous pouvez mettre des pruneaux (ou autre chose). Le reste de la recette est identique, le résultat **dépend uniquement de la valeur de la condition**.

Boucle **For**: Dans une recette de cuisine, vous n'avez jamais vu "cassez un œuf, puis un deuxième œuf, puis un troisième œuf, puis un quatrième œuf..." (ça peut durer longtemps). Il est indiqué "cassez X œufs".
En algorithmique, on utilise la boucle **For** qui permet de **répéter plusieurs fois la même opération** : Comptez de (For i=) 1 jusqu'à X, à chaque fois, cassez un œuf. (Attention, c'est plus beaucoup plus subtil que ça, en particulier, **l'opération exécutée lors d'une itération i peut dépendre de là où vous en êtes dans votre décompte, et des itérations précédentes**)

Boucle **While**: Cette boucle sert à réaliser les opérations du genre "mélanger jusqu'à ce que la pâte soit lisse " (autrement dit, mélangez la pâte **tant que** celle-ci est pleine de grumeaux...").
Cela permet d'exprimer des instructions du genre : "mélanger la pâte pendant 2 min., si il y a encore des grumeaux, mélanger la pâte 2 min., si il y a encore des grumeaux, mélanger la pâte 2 min, si il y a encore des grumeaux, mélanger la pâte..." (encore une fois, ça peut durer longtemps).
Là aussi, la **condition d'arrêt** peut varier selon **l'itération** précédente.

Bases algorithmiques IV : Boucles Conditionnelles

IF ... THEN ... ELSE ....

Une **proposition bouléenne** (dépendant des entrées de la fonction ou de calculs préalables) est évaluée. Selon sa valeur, la fonction fait telle ou telle chose.

```
>       Caml Light version 0.80

#let test n =
        if ((n mod 2) = 0)
                then print_string "n est pair"
                else print_string "n est impair";;
test : int -> unit = <fun>
#test 8;;
n est pair- : unit = ()
#test 17;;
n est impair- : unit = ()
#
```

(x mod y) est une fonction qui renvoie le reste de la division euclidienne de x par y

IF Condition THEN ... ELSE....

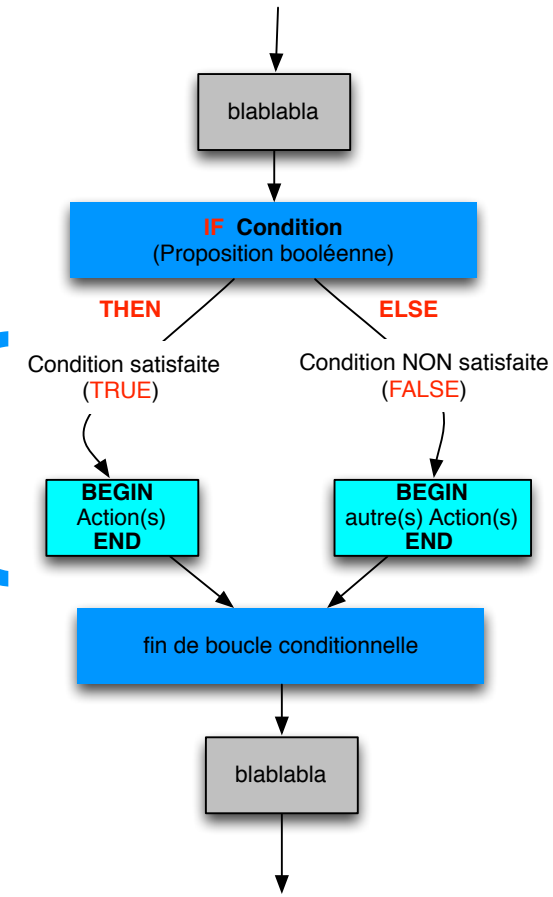
"print_string Y" affiche/écrit dans la console la chaîne de caractère Y

```
>       Caml Light version 0.80

#let pari n=
        let alea = random__int 100 in
        if (n>alea) then print_string "Gagné" else print_string "Perdu";;
pari : int -> unit = <fun>
#pari 56;;
Gagné- : unit = ()
#pari 3;;
Perdu- : unit = ()
#pari 500;;
Gagné- : unit = ()
```

"random__int x" renvoie un entier aléatoire entre 0 et x-1

IF Condition THEN... ELSE...

Boucle Conditionnelle



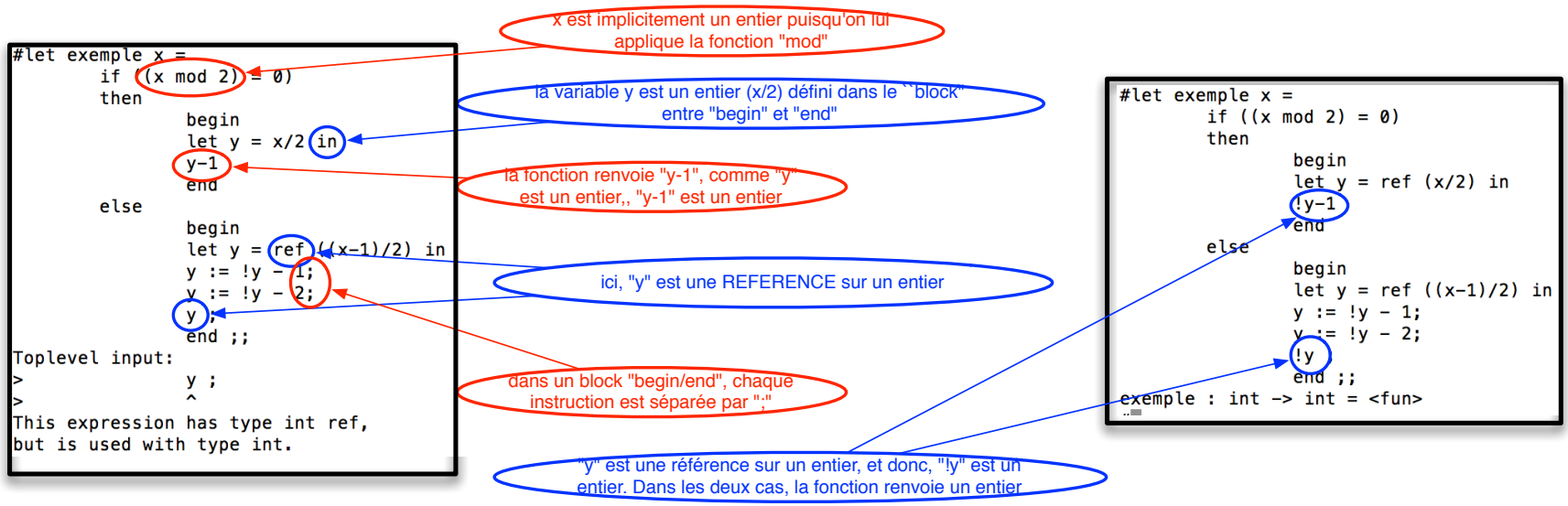
Que font les fonctions "test" et "pari" ci-dessus ?

Si plusieurs instructions sont à réaliser dans "then" ou "else", elles sont “encadrées" par **begin** et **end**.

Bases algorithmiques IV : Boucles Conditionnelles

Court (et informel) aparté technique pour rappeler qu'**en CAML, tout est typé**.

En particulier, **le type renvoyé par une fonction est unique**. I.e., dans une boucle conditionnelle, faîtes attention à bien renvoyer le même type dans tous les cas



Bases algorithmiques IV : Boucles Conditionnelles

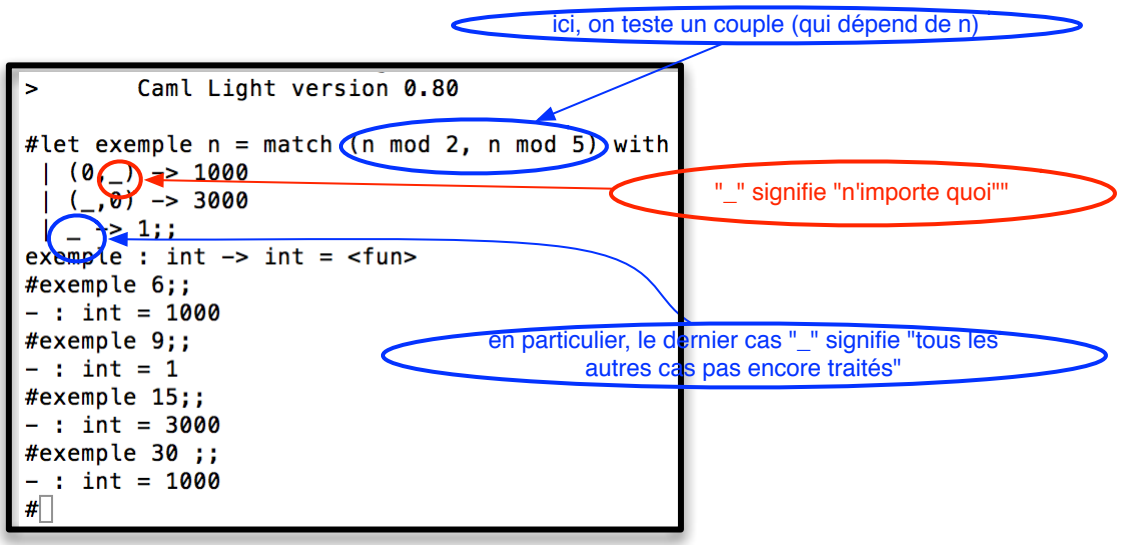
MATCH ... WITH ...

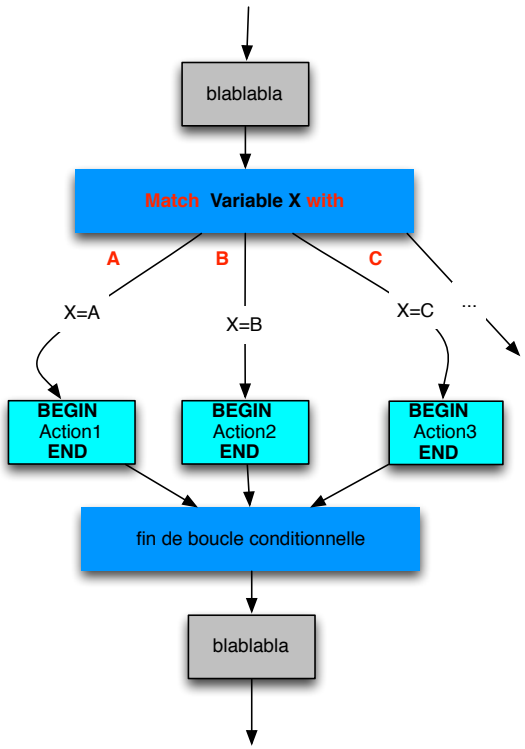
En CAML, **if...then...else...** n'est pas la seule option pour les boucles conditionnelles.

match x with |A → action1 |B → action2 |C → action3 ... permet d'exécuter des actions différentes selon la valeur de x.

Cela revient à faire **if x=A then action1 else if x=B then action2 else if x=C then action3...**

Notez que l'ordre est primordial ! si (x=A) ET (x=C) sont vraies, alors seule action1 est réalisée.





Que fait la fonction "exemple" ci-dessus ?

Lors d'un "match X with", il est également primordial d'être sur que tous les cas (toutes les valeurs possibles pour X) sont traités.

Bases algorithmiques IV : Boucle FOR : "Motivations"

Écrivez un algorithme qui calcule (renvoie/retourne) $sin^2(1) + sin^2(2) + sin^2(3) + sin^2(4)$.

Voici 2 possibilités avec les outils dont nous disposons jusqu'à présent :

Comment **généraliser** ces algorithmes pour calculer la somme des carrés des sinus des n premiers entiers ? *Écrivez un algorithme qui, étant donné n, calcule* $\sum_{i=1}^{n} sin^2(i)$

Ce n'est *a priori* pas possible uniquement avec les outils présentés jusqu'à présent !

L'exemple de droite semble (peut-être) plus compliqué, mais en fait il ne fait que répéter **systématiquement** (4 fois) la "même" opération. C'est cet algorithme que la notion de boucle "FOR" va nous permettre de généraliser.

Bases algorithmiques IV : Boucle FOR : "Motivations"

Écrivez un algorithme qui calcule (renvoie/retourne) $sin^2(1) + sin^2(2) + sin^2(3) + sin^2(4)$.

Voici 2 possibilités avec les outils dont nous disposons jusqu'à présent :

```
>       Caml Light version 0.80

#sin 4. ;;
- : float = -0.756802495308
#sin 1. ;;
- : float = 0.841470984808
#sin 3.14156 ;;
- : float = 3.26535897873e-05
#3. ** 2. ;;
- : float = 9.0
#4. ** 2. ;;
- : float = 16.0
#(sin 4.) ** 2. ;;
- : float = 0.572750016904
#let som4firstSinusCarre =
        ((sin 1.) ** 2.) +. ((sin 2.) ** 2.)
 +. ((sin 3.) ** 2.) +. ((sin 4.) ** 2.);;
som4firstSinusCarre : float = 2.12756010228
#let x = som4firstSinusCarre;;
x : float = 2.12756010228
#x;;
- : float = 2.12756010228
#
```

"sin x" est une fonction qui renvoie le sinus (un flottant) d'un flottant "x", ici 4. (notez le "." qui signifie qu'on a un flottant)

sinus de "presque" Pi vaut "presque 0

"x ** y" est une fonction qui prend 2 flottants x et y en entrée et renvoie le flottant x puissance y

Notez qu'une fonction peut n'avoir aucune entrée
Notez aussi, qu'ici les additions sont "+." (Notez le ".") puisqu'on additionne des flottants

```
>       Caml Light version 0.80

#let som4firstSinusCarreV2 =
        let somme_courante = ref 0.0 in
        somme_courante := !somme_courante +. ((sin 1.) ** 2.);
        somme_courante := !somme_courante +. ((sin 2.) ** 2.);
        somme_courante := !somme_courante +. ((sin 3.) ** 2.);
        somme_courante := !somme_courante +. ((sin 4.) ** 2.);
        !somme_courante;;
som4firstSinusCarreV2 : float = 2.12756010228
#let x = som4firstSinusCarreV2;;
x : float = 2.12756010228
#x;;
- : float = 2.12756010228
#
```

cette fonction crée une variable (référence), appelée "somme_courante", initialisée au flottant 0.
et y ajoute itérativement (sin 1)*(sin 1), puis (sin 2)*(sin 2) ... puis (sin 4)*(sin 4)
finalement, elle renvoie la valeur de "somme_courante"

Comment **généraliser** ces algorithmes pour calculer la somme des carrés des sinus des n premiers entiers ? *Écrivez un algorithme qui, étant donné n, calcule* $\sum_{i=1}^{n} sin^2(i)$

Ce n'est *a priori* pas possible uniquement avec les outils présentés jusqu'à présent !

L'exemple de droite semble (peut-être) plus compliqué, mais en fait il ne fait que répéter **systématiquement** (4 fois) la "même" opération. C'est cet algorithme que la notion de boucle "FOR" va nous permettre de généraliser.

Bases algorithmiques IV : Boucle FOR : "Motivations"

Écrivez un algorithme qui calcule (renvoie/retourne) $sin^2(1) + sin^2(2) + sin^2(3) + sin^2(4)$.

Voici 2 possibilités avec les outils dont nous disposons jusqu'à présent :

```
>       Caml Light version 0.80

#sin 4. ;;
- : float = -0.756802495308
#sin 1. ;;
- : float = 0.841470984808
#sin 3.14156 ;;
- : float = 3.26535897873e-05
#3. ** 2. ;;
- : float = 9.0
#4. ** 2. ;;
- : float = 16.0
#(sin 4.) ** 2. ;;
- : float = 0.572750016904
#let som4firstSinusCarre =
        ((sin 1.) ** 2.) +. ((sin 2.) ** 2.)
 +. ((sin 3.) ** 2.) +. ((sin 4.) ** 2.);;
som4firstSinusCarre : float = 2.12756010228
#let x = som4firstSinusCarre;;
x : float = 2.12756010228
#x;;
- : float = 2.12756010228
#
```

"sin x" est une fonction qui renvoie le sinus (un flottant) d'un flottant "x", ici 4. (notez le "." qui signifie qu'on a un flottant)

sinus de "presque" Pi vaut "presque 0

"x ** y" est une fonction qui prend 2 flottants x et y en entrée et renvoie le flottant x puissance y

Notez qu'une fonction peut n'avoir aucune entrée
Notez aussi, qu'ici les additions sont "+." (Notez le ".") puisqu'on additionne des flottants

```
>       Caml Light version 0.80

#let som4firstSinusCarreV2 =
        let somme_courante = ref 0.0 in
        somme_courante := !somme_courante +. ((sin 1.) ** 2.);
        somme_courante := !somme_courante +. ((sin 2.) ** 2.);
        somme_courante := !somme_courante +. ((sin 3.) ** 2.);
        somme_courante := !somme_courante +. ((sin 4.) ** 2.);
        !somme_courante;;
som4firstSinusCarreV2 : float = 2.12756010228
#let x = som4firstSinusCarreV2;;
x : float = 2.12756010228
#x;;
- : float = 2.12756010228
#
```

cette fonction crée une variable (référence), appelée "somme_courante", initialisée au flottant 0.
et y ajoute itérativement (sin 1)*(sin 1), puis (sin 2)*(sin 2) ... puis (sin 4)*(sin 4)
finalement, elle renvoie la valeur de "somme_courante"

Comment **généraliser** ces algorithmes pour calculer la somme des carrés des sinus des n premiers entiers ? *Écrivez un algorithme qui, étant donné n, calcule* $\sum_{i=1}^{n} sin^2(i)$

Ce n'est *a priori* pas possible uniquement avec les outils présentés jusqu'à présent !

L'exemple de droite semble (peut-être) plus compliqué, mais en fait il ne fait que répéter **systématiquement** (4 fois) la "même" opération. C'est cet algorithme que la notion de boucle "FOR" va nous permettre de généraliser.

Bases algorithmiques IV : Boucle FOR : "Motivations"

Écrivez un algorithme qui calcule (renvoie/retourne) $sin^2(1) + sin^2(2) + sin^2(3) + sin^2(4)$.

Voici 2 possibilités avec les outils dont nous disposons jusqu'à présent :

```
>       Caml Light version 0.80

#sin 4. ;;
- : float = -0.756802495308
#sin 1. ;;
- : float = 0.841470984808
#sin 3.14156 ;;
- : float = 3.26535897873e-05
#3. ** 2. ;;
- : float = 9.0
#4. ** 2. ;;
- : float = 16.0
#(sin 4.) ** 2. ;;
- : float = 0.572750016904
#let som4firstSinusCarre =
        ((sin 1.) ** 2.) +. ((sin 2.) ** 2.)
 +. ((sin 3.) ** 2.) +. ((sin 4.) ** 2.);;
som4firstSinusCarre : float = 2.12756010228
#let x = som4firstSinusCarre;;
x : float = 2.12756010228
#x;;
- : float = 2.12756010228
#
```

"sin x" est une fonction qui renvoie le sinus (un flottant) d'un flottant "x", ici 4. (notez le "." qui signifie qu'on a un flottant)

sinus de "presque" Pi vaut "presque 0

"x ** y" est une fonction qui prend 2 flottants x et y en entrée et renvoie le flottant x puissance y

Notez qu'une fonction peut n'avoir aucune entrée
Notez aussi, qu'ici les additions sont "+." (Notez le ".") puisqu'on additionne des flottants

```
>       Caml Light version 0.80

#let som4firstSinusCarreV2 =
        let somme_courante = ref 0.0 in
        somme_courante := !somme_courante +. ((sin 1.) ** 2.);
        somme_courante := !somme_courante +. ((sin 2.) ** 2.);
        somme_courante := !somme_courante +. ((sin 3.) ** 2.);
        somme_courante := !somme_courante +. ((sin 4.) ** 2.);
        !somme_courante;;
som4firstSinusCarreV2 : float = 2.12756010228
#let x = som4firstSinusCarreV2;;
x : float = 2.12756010228
#x;;
- : float = 2.12756010228
#
```

cette fonction crée une variable (référence), appelée "somme_courante", initialisée au flottant 0.
et y ajoute itérativement (sin 1)*(sin 1), puis (sin 2)*(sin 2) ... puis (sin 4)*(sin 4)
finalement, elle renvoie la valeur de "somme_courante"

Comment **généraliser** ces algorithmes pour calculer la somme des carrés des sinus des n premiers entiers ? *Écrivez un algorithme qui, étant donné n, calcule* $\sum_{i=1}^{n} sin^2(i)$

Ce n'est *a priori* pas possible uniquement avec les outils présentés jusqu'à présent !

L'exemple de droite semble (peut-être) plus compliqué, mais en fait il ne fait que répéter **systématiquement** (4 fois) la "même" opération. C'est cet algorithme que la notion de boucle "FOR" va nous permettre de généraliser.

Bases algorithmiques IV : principes de la Boucle FOR

FOR $i = a$ TO b DO $action(i, \cdots)$ DONE;

Une boucle **for** permet de répéter (**itérer**) plusieurs fois la "même" opération.

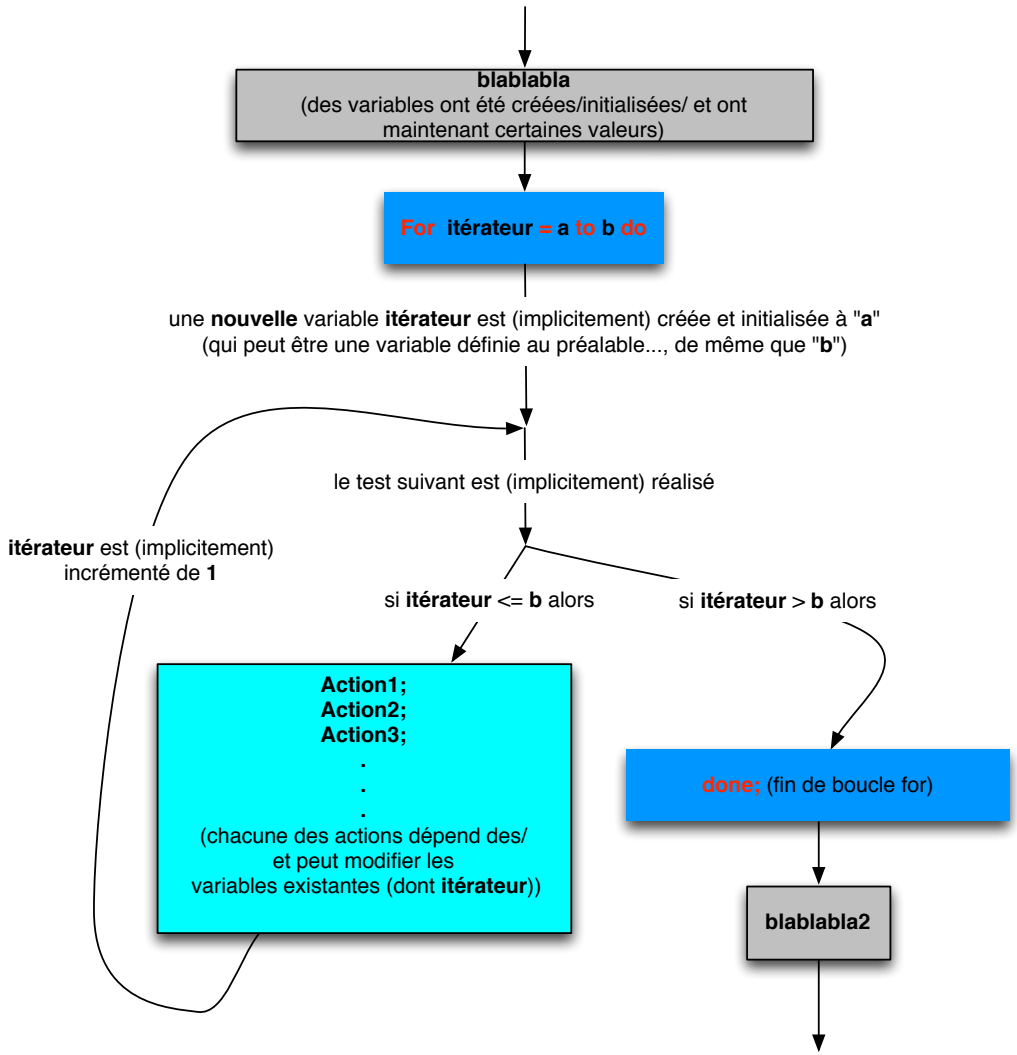
À chaque **itération**, l'opération réalisée dépend cependant de la valeur de l'**itérateur** (ou **compteur**) et des variables courantes.

Ex: Calcul de $\sum_{i=1}^{n} sin^2(i)$: "somme des $sin(i)^2$ pour i allant de 1 à n"

```
>       Caml Light version 0.80

#let somFirstSinusCarre n =
        let somme_courante = ref 0. in
        for i=1 to n do
                somme_courante := !somme_courante +. ((sin (float_of_int i)) ** 2.)
        done;
        !somme_courante;;
somFirstSinusCarre : int -> float = <fun>
#let x = somFirstSinusCarre 4;;
x : float = 2.12756010228
#x;;
- : float = 2.12756010228
#
```

dans une boucle FOR,
l'itérateur (ici "i") est un ENTIER
dans cet exemple, on veut l'utiliser avec une fonction qui demande un flottant.
On utilise la fonction "float_of_int" pour cela

Apparté : Notons (on y reviendra) qu'il faut s'assurer que l'itérateur atteint une valeur $> b$ après un nombre fini d'itérations, **sinon le programme boucle indéfiniment**



Bases algorithmiques IV : Boucle FOR :

des exemples simples à comprendre et connaître !

Somme des *n* premiers entiers

Ce calcul peut se faire sans boucle For.

Voyez vous comment ? (on y reviendra)

```
>       Caml Light version 0.80

#let somme n =
        let somme_courante = ref 0 in
        for i=1 to n do
                somme_courante := !somme_courante + i
        done;
        !somme_courante;;
somme : int -> int = <fun>
#somme 20;;
- : int = 210
#
```

```
>       Caml Light version 0.80

#let Algo_tab =
        let n = vect_length tab in
        let sol_courante = ref tab.(0) in
        for curseur = 1 to (n-1) do
                if tab.(curseur)> !sol_courante
                        then sol_courante := tab.(curseur)
        done;
        !sol_courante;;
Algo : 'a vect -> 'a = <fun>
#Algo [|7;2;9;5|];;
- : int = 9
#
```

pour travailler sur un tableau, il faut en connaître la taille

Rappel : un tableau de longueur n est indice de 0 à n-1

Une boucle FOR est très utile pour parcourir des structures de données comme les tableaux

Que fait l'algorithme `Algo` ? l'algorithme `Algo2` ?

```
>       Caml Light version 0.80

#let matrice = [| [|5;6;7|] ; [|12;54;2|] ; [|1;1;1|] ; [|2;3;4|] |];;
matrice : int vect vect =
 [|[|5; 6; 7|]; [|12; 54; 2|]; [|1; 1; 1|]; [|2; 3; 4|]|]
#matrice.(1);;
- : int vect = [|12; 54; 2|]
#matrice.(3).(1);;
- : int = 3
#vect_length matrice ;;
- : int = 4
#vect_length matrice.(0) ;;
- : int = 3
#let Algo2 mat =
        let n = vect_length mat in
        let m = vect_length mat.(0) in
        let sol_courante = ref 0 in
        for i = 0 to (n-1) do
                for j = 0 to (m-1) do
                        sol_courante := !sol_courante + mat.(i).(j)
                done;
        done;
        !sol_courante;;
Algo2 : int vect vect -> int = <fun>
#Algo2 matrice;;
- : int = 98
#
```

en CAML, on peut définir une matrice comme un tableau de tableaux (de même longueurs)

corps de la première boucle for

Boucles imbriquées

Dans le corps d'une boucle For, on peut

bien sur en mettre une autre

Bases algorithmiques IV : Boucle WHILE

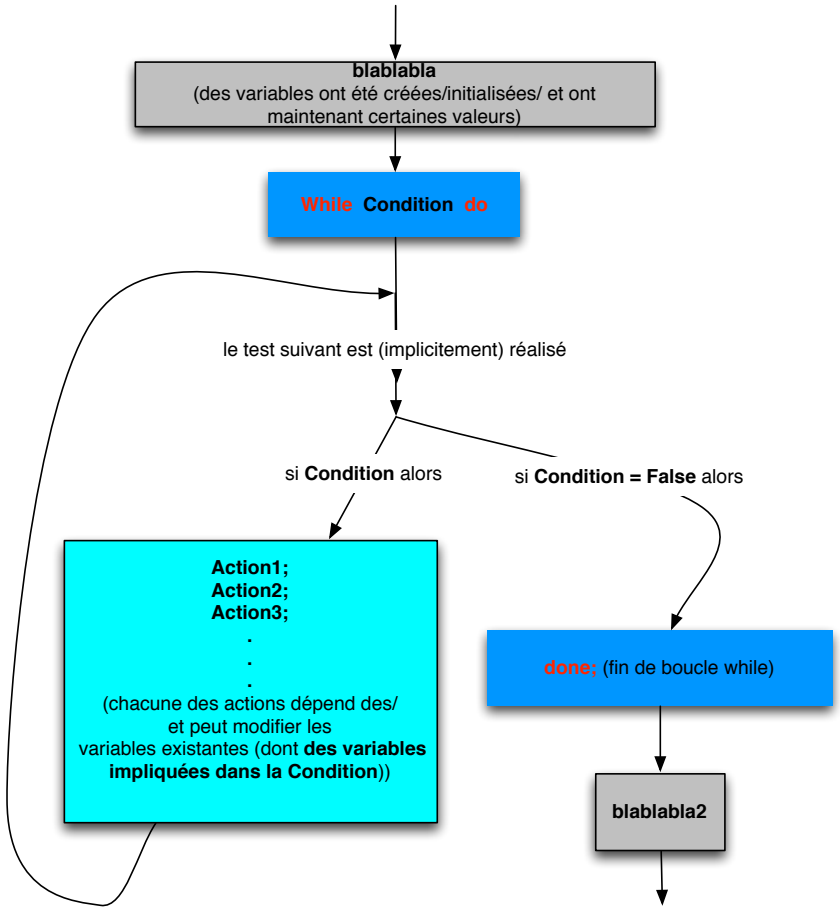
Lorsqu'on veut répéter des opérations, mais qu'on ne connaît *a priori* pas le nombre d'itérations (on s'arrête ou continue en fonction de l'état courant), alors la boucle WHILE est utile

Ex : Suite de Syracuse

Soit la suite $(u_n)_{n\in\mathbb{N}}$ définie par $u_0 \in \mathbb{N}^*$
et $u_{n+1} = u_n/2$ si n pair et $u_{n+1} = 3 * u_n + 1$ sinon.

Conjecture: cette suite atteint toujours 1 (quel que soit u_0) et donc cycle $(1, 4, 2, 1...)$

```
#let Syracuse n =
        let nombre_iteration = ref 0 in
        let valeur_courante = ref n in
        while !valeur_courante >1 do
                if (!valeur_courante mod 2) = 0
                        then valeur_courante := !valeur_courante/2
                        else valeur_courante := !valeur_courante*3+1;
                nombre_iteration := !nombre_iteration +1;
        done;
        !nombre_iteration;;
Syracuse : int -> int = <fun>
#Syracuse 14;;
- : int = 17
```



Une boucle WHILE peut "simuler" une boucle FOR

Comparer avec `Algo` au slide précédent

```
#let Algo3 tab =
        let n = vect_length tab in
        let sol_courante = ref tab.(0) in
        let i = ref 1 in
        while !i<n do
                if tab.(!i)> !sol_courante
                        then sol_courante := tab.(!i)
                        else ();
                i := !i +1;
        done;
        !sol_courante;;
Algo3 : 'a vect -> 'a = <fun>
#
```

Outline

Bases algorithmiques V : Algorithmes récursifs

À mon avis, il s'agit du point le plus compliqué et le plus important du programme

Une fonction récursive est une fonction qui s'appelle elle-même

En CAML, on utilise le mot "rec" : `let rec nom_fonction =...`

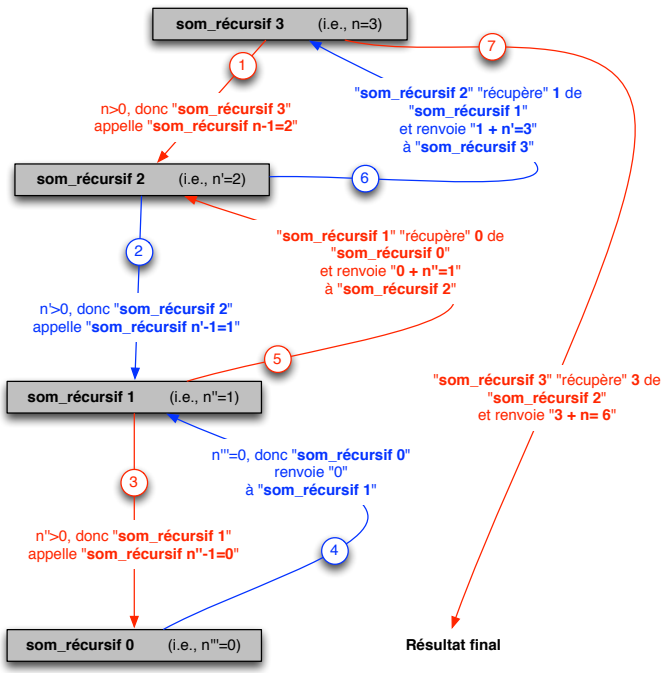
Ce slide ne présente qu'une première approche (très rapide et "simple") avec notre exemple préféré :

Somme des n premiers entiers : récursif

```
#let rec som_recursif n =
        match n with
        |0 -> 0
        |_ -> n + (som_recursif (n-1));;
som_recursif : int -> int = <fun>
#
```

Cet algorithme se lit : "`som_recursif n` renvoie 0 si $n = 0$, et renvoie n plus la valeur que renvoie `som_recursif n-1` sinon"

Faîtes le parallèle avec l'expression de $u_n = \sum_{i=0}^{n} i$ sous forme de suite de récurrence.
$u_0 = 0$ et $u_n = n + u_{n-1}$ pour tout $n > 0$.



Il est important (crucial ?) de comprendre la succession (*pile*) des appels récursifs lors de l'éxécution d'une fonction récursive

Comme pour une suite récurrente, pour définir un algorithme récursif, il faudra TOUJOURS bien définir le cas de base (dans cet exemple $n = 0$) ... on va y revenir