Projet programmation fonctionnelle : Introduction

Julien Narboux

19 septembre 2024

Présentation de Ocaml

Ocaml est un langage :

- multi-paradigmes,
- interprété et compilé,
- les fonctions sont des objets de première classe
- avec gestion automatique de la mémoire
- à typage statique,
- supportant le polymorphisme paramétrique
- avec inférence de type
- offrant un mécanisme de filtrage
- supportant les exceptions
- ainsi que les types algébriques de données

La programmation fonctionnelle et l'industrie

Garbage collection Java [1995], LISP [1958]

Generics Java 5 [2004], ML [1990]

Fonctions d'ordre supérieur C# 3.0 [2007], Java 8 [2014], LISP [1958]

Inférence de types C++11 [2011], Java 7 [2011] and 8, ML

Filtrage de motifs Python 3.10 [2020]

Objectifs de introduction

- Connaître les grands *paradigmes* de programmation.
- Connaître des éléments de vocabulaire pour décrire les langages de programmation.
- Afin de vous préparer à pouvoir aborder *plusieurs* langages.

Turing complétude

D'un point de vue théorique ils sont tous équivalents !

Turing complétude

D'un point de vue théorique ils sont tous équivalents !

Définition

Un dit qu'un langage est Turing complet, quand il permet de programmer n'importe quelle fonction calculable au sens de Church-Turing (voir Bloc 5).

Turing complétude

D'un point de vue théorique ils sont tous équivalents !

Définition

Un dit qu'un langage est Turing complet, quand il permet de programmer n'importe quelle fonction calculable au sens de Church-Turing (voir Bloc 5).

Exemples :

Python, C, C++, Java, Lisp, OCaml, Latex, ...

Turing complétude

D'un point de vue théorique ils sont tous équivalents !

Définition

Un dit qu'un langage est Turing complet, quand il permet de programmer n'importe quelle fonction calculable au sens de Church-Turing (voir Bloc 5).

Exemples :

Python, C, C++, Java, Lisp, OCaml, Latex, ...

Contre-exemples :

XML, HTML, automates, Coq...

Turing complétude

D'un point de vue théorique ils sont tous équivalents !

Définition

Un dit qu'un langage est Turing complet, quand il permet de programmer n'importe quelle fonction calculable au sens de Church-Turing (voir Bloc 5).

Exemples :

Python, C, C++, Java, Lisp, OCaml, Latex, ...

Contre-exemples :

XML, HTML, automates, Coq...

Turing complets par accident :

HTML+CSS, Templates C++, XSLT, ...

Pouvoir d'expressivité

langage programmation $\neq$ langage mathématique

- visée opérationnelle
- compromis entre *puissance d'expression* et *capacité d'exécution*

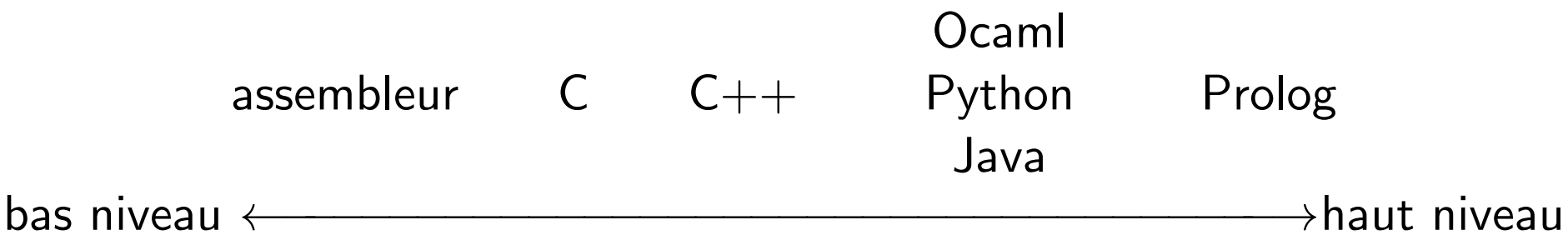
Langage de haut/bas niveau

langage de haut niveau

- Les calculs sont décrits de manière plus abstraite.
- ☺Programmation plus facile : le programmeur ne doit pas gérer tous les détails.
- ☹Maîtrise des détails plus faible.

langage de bas niveau

- Les calculs sont décrits dans un langage proche de la machine réelle.
- ☹Programmation plus difficile.
- ☺Maîtrise des détails plus grande.



Exemple d'un programme de haut niveau

- La liste vide est triée.
- Toute liste singleton est triée.
- Si X est plus petit que Y et que la liste Y : :T est triée alors la liste X : :Y : :T est triée.
- Si L est une permutation de S et si S est triée alors S est le résultat d'un tri de L.

Exemple d'un programme de haut niveau

- La liste vide est triée.
- Toute liste singleton est triée.
- Si X est plus petit que Y et que la liste Y : :T est triée alors la liste X : :Y : :T est triée.
- Si L est une permutation de S et si S est triée alors S est le résultat d'un tri de L.

Prolog

```
naive_sort(List,Sorted):-
        perm(List,Sorted),is_sorted(Sorted).

is_sorted([]).
is_sorted)[_]).
is_sorted([X,Y|T]):-X=<Y,is_sorted([Y|T]).
```

Syntaxe et sémantique

Exemples I

En C

```c
#include <stdio.h>
#include <stdlib.h>

void toto() {
  int i;
  for (i=0;i<20;i++)
    printf("%d\n",i);
}

int main() {
  toto();
}
```

Exemples II

En C

```c
#include <stdio.h>
#include <stdlib.h>

void toto() {
  int i;
  for (i=0;i<0x14;i++)
    printf("%d\n",i);
}

int main() {
  toto();
}
```

Exemples III

En Ocaml

```
open Printf

let toto () =
  for i = 0 to 19 do
    printf "%d\n" i;
  done
;;

let _ = toto()
```

A l'université, on ne s'intéresse pas à la syntaxe.

Pouvoir d'expressivité

On dit qu'une fonctionnalité F d'un langage augmente le pouvoir d'expressivité du langage si on ne peut passer d'un programme utilisant F à un programme n'utilisant pas F par un changement purement local. [1]

Exemple Les exceptions

Contre-exemple Sucre syntaxique, par exemple `-x` peut être vu comme du sucre syntaxique pour `0-x`

1. Felleisen, On the expressive power of programming languages, 1990

Quelques paradigmes

Un paradigme est une manière de programmer un ordinateur basé sur des *principes* ou un point de vue.

- Programmation impérative
- Programmation structurée
- Programmation fonctionnelle
- Programmation orientée objet
- Programmation déclarative
- Programmation concurrente
- Programmation distribuée
- ...

Plan

Programmation impérative

- Programme caractérisé pas un état implicite (la mémoire) qui est modifié par des instructions.
- Paradigme historique, car le plus proche du fonctionnement de la machine.

Exemples :

Assembleur, Algol, Fortran, Cobol, Basic, Pascal, C, Java, Ocaml,...

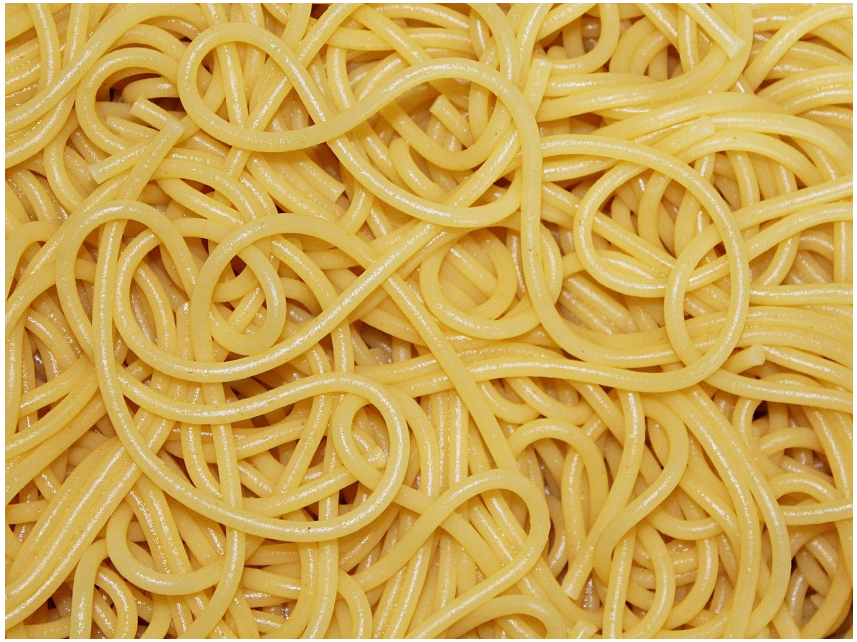
Programmation impérative

- Séquence d'instructions : $i_1; i_2$
- Affectation : $x := expr$
- Instruction conditionnelle : if e then i else$i$
- Saut inconditionnel *goto l*

Plan

Programmation structurée

- On s'interdit le `goto`.
- On utilise des conditionnelles plus structurées (switch), des boucles (for, while, ...), blocs, routines
- Wirth : Algol W (1970)



Code Spaghetti

Exemples :

Algol W, Pascal, Ada, Ocaml, Java, C, ...

Contre-exemples :

assembleur, basic, ...

Programmation structurée

- On s'interdit le `goto`.
- On utilise des conditionnelles plus structurées (switch), des boucles (for, while, ...), blocs, routines
- Wirth : Algol W (1970)

Code structuré

Exemples :

Algol W, Pascal, Ada, Ocaml, Java, C, ...

Contre-exemples :

assembleur, basic, ...

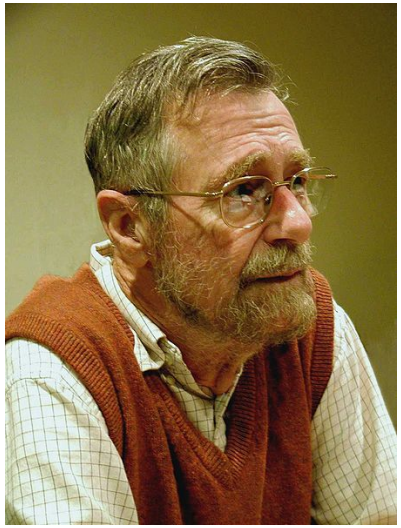
Exemple

En C

```c
void printNumbers()
{
    int n = 1;
label:
    printf("%d ",n);
    n++;
    if (n <= 10)
        goto label;
}
```

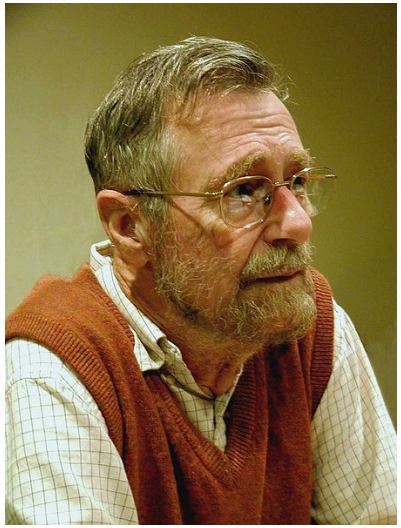
E.W. DIJKSTRA "Goto Statement Considered Harmful."

Ce paradigme fut popularisé par l'article de Dijkstra : "Goto Statement Considered Harmful", 1968



E.W. DIJKSTRA "Goto Statement Considered Harmful."

Ce paradigme fut popularisé par l'article de Dijkstra : "Goto Statement Considered Harmful", 1968



"Please don't fall into the trap of believing that I am terribly dogmatic about [the go to statement]. I have the uncomfortable feeling that others are making a religion out of it, as if the conceptual problems of programming could be solved by a simple trick, by a simple form of coding discipline !"

Dijkstra

A propos des break et des return

Certains se donnent comme principe d'écrire des boucles ou des fonctions qui n'ont qu'une seule *sortie*. Cette contrainte impose d'utiliser des variables supplémentaires.
Roberts argumente que cela rend les programmes plus difficiles à écrire. [2]

2. Eric S. Roberts, Loop Exits and Structured Programming : Reopening the Debate

Exemple

```
def recherche2(x, l):
    i = 0
    while i < len(l):
        if x == l[i]:
            return i
        i = i + 1
    return -1
```

Exercice

Proposer une solution sans return dans la boucle.

Exercice

Proposer une solution sans return dans la boucle.

```
def recherche(x,l):
    i=0
    trouve = False
    indice = -1
    while (i<len(l) and not trouve):
        if x == l[i]:
            trouve=True
            indice=i
        i= i+1
    return indice
```

Quelques principes de découpage fonctionnel

- Éviter de mélanger calcul et affichage/saisie.
- Une fonction ne doit pas dépendre de variables globales
- Une fonction ne doit pas avoir trop de paramètres, on peut regrouper les paramètres avec un type produit.
- Une fonction ne doit pas être trop longue, sinon découper en sous-fonctions.
- Une fonction ne doit pas avoir trop de structures de contrôle imbriquées.
- Si du code similaire apparaît plusieurs fois, c'est qu'il fallait faire une fonction.

Plan

Programmation orientée objet

- Assemblage de briques logicielle appelées *objet*
- Objets = attributs + méthodes = état + fonctions
- Notion de classe : une catégorie d'objets
- Notion de hiérarchie, généralisation, spécialisation.

Exemples :

Smalltalk (1972), C++, Java, C#, PHP, Ada, Objective C, Ocaml, Python,...

Exemple en C++

```
class Point
{
  int x;
  int y;

public:
  Point(int x, int y) : x(x), y(y) {}
  int getX() const { return x; }
  int getY() const { return y; }
  bool isOrigin() const { return x == 0 && y == 0; }
  Point translate(const Point& point) const {
    return Point(x + point.x, y + point.y);
  }
};
```

Source : Wikipedia

Exemple en Java

```java
public class Point {
  private final int x;
  private final int y;

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public int getX() { return x; }
  public int getY() { return y; }
  public boolean isOrigin() { return (x == 0) && (y == 0); }
  public Point translate(Point point) {
    return new Point(x + point.x, y + point.y);
  }
}
```

En Python

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def translate(self, dx, dy):
        self.x = self.x + dx
        self.y = self.y + dy

    def funtrans(self, dx, dy):
        return Point(self.x + dx, self.y + dy)
```

Encapsulation

Deux idées :

1. regrouper les données et les fonctions permettant de les manipuler
2. restreindre l'accès à des données

Motivations :

- Modularité
- Pouvoir changer la structure de données sans changer l'interface.

Restriction d'accès

En Python :

```
class Point:
    def __init__(self, x, y):
        self.__x = x
        self.__y = y
```

Dans d'autres langages, on a : private, public, protected

Plan

Programmation déclarative

- On décrit le 'quoi' plutôt que le 'comment'
  1. Programmation fonctionnelle
  2. Programmation logique
     Exemples : Prolog,. . .
  3. Programmation par contraintes
     Exemples : Oz, . . .

Plan

Programmation fonctionnelle

- Programmer uniquement avec des fonctions.
- Pas d'affectation.
- Programmation sans état.
- Pas de mise à jour des structures de données.
- Pas d'effets de bord.
- Les fonctions sont des objets de première classe (on peut prendre en argument des fonctions et on peut renvoyer une fonction).

Exemples :

Lisp, Scheme, ML, Haskell, Erlang, Ocaml, Scala, F#, ...

Programmation fonctionnelle

- Les répétitions sont exprimées avec la récursivité
- La séquence est remplacée par la composition de fonctions

Quelques fonctions incontournables sur les listes

1. filter
2. map
3. fold/reduce

Map

La fonction Map permet d'appliquer la même fonction à tous les éléments d'une liste.
La fonction Filter permet de sélectionner les éléments d'une liste vérifiant une condition donnée.

En Python

```
items = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, items))
```

Filter

La fonction Filter permet de sélectionner les éléments d'une liste vérifiant une condition donnée.

En Python

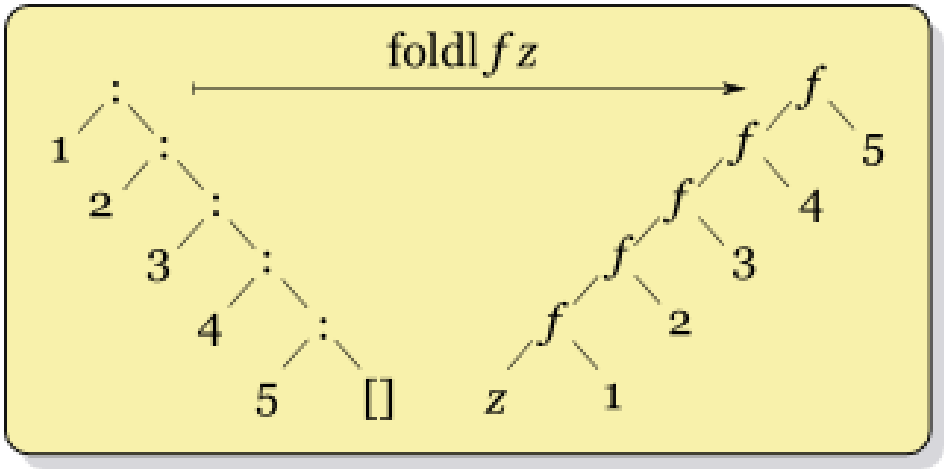
```
number_list = range(-5, 5)
less_than_zero = list(filter(lambda x: x < 0, number_list))
```

Schéma de compréhension

En Python

```
[f(x) for x in l]
[x for x in l if p(x)]
```

Fold/Reduce



Intuitivement, la fonction `fold` consiste à insérer un opérateur binaire entre chaque paire d'éléments consécutifs d'une liste.

Reduce/Fold

En Python

```
from functools import reduce
product = reduce((lambda x, y: x * y), [1, 2, 3, 4])
```

Reduce/Fold

En Ocaml

```
let product l = List.fold_left (fun x y -> x*y) 1 l;;
```

Plan

Programmation logique

- Un programme est composé d'un ensemble de faits et règles.
- Le programmeur ne donne pas de description des calculs à réaliser.

Plan

Programmation événementielle

Le déroulement du programme est dicté par les événements qui se déclenchent. Le programmeur décide de ce qui se passe quand un événement se produit. Les événements peuvent être des messages reçus, des cliques de souris, la mise à jour d'un champs de texte, ...

Plan

Programmation concurrente

Plusieurs processus en même temps.

Exemples :

Erlang, JoinCaml,. . .

Plan

Programmation distribuée

Sur plusieurs ordinateurs en même temps.

Exemples :

Erlang,. . .

Plan

Langage compilé/interprété

Programme compilé

Le programme est transformé en un programme en code machine directement exécutable par le microprocesseur.

Exemples

C, C++, Fortran, Ocaml,. . .

Programme interprété

Un programme (appelé interprète ou machine virtuelle) exécute le programme.

- ☹lenteur.
- ☺portabilité

Exemples

Java, Javascript, PHP, Perl, Ocaml, Ruby, Python, . . .

Bytecode

Certains langages permettent de compiler le programme sous forme de *bytecode*. Un *bytecode* est un code intermédiaire de bas niveau mais qui n'est pas exécutable directement par un microprocesseur.

Exemples

Java, Ocaml, Python, PHP, Perl, ...

Just in Time

Programme interprété mais certaines parties sont compilées à la volée.

Exemples

Smalltalk, Java, Javascript (Firefox $>$ 3.1), ...

Plan

Types

Définition

Le type d'un objet est une classe identifiant les différentes valeurs que cette objet peut prendre, les opérations qu'on peut lui appliquer et la façon dont les données sont stockées.

On distingue :

- les types primitifs (int, float, string, char, bool, ...)
- les types fonctionnels
- ...

Exemple

En Ocaml

```
# let x = 3;;
val x : int = 3
# let y = 3.0;;
val y : float = 3.
# let f x = x+2;;
val f : int -> int = <fun>
# let g x = x +. 2.0;;
val g : float -> float = <fun>
```

En Python

```
>>> x = 3
>>> type(x)
<type 'int'>
>>> y=3.0
>>> type(y)
<type 'float'>
def f(x):
  return (x+2)

type(f)
<type 'function'>
```

Typage

Empêcher des opérations impossibles :

```
1+"toto"
```

Typage statique/dynamique

Typage statique

Le typage est calculé à la compilation.

Typage dynamique

Le typage est vérifié à l'exécution.

Exemples

C, Ocaml, Haskell, Java, ...

Exemples

Python, PHP, Javascript, Java, ...

Typage statique/dynamique

Typage statique

Le typage est calculé à la compilation.

Typage dynamique

Le typage est vérifié à l'exécution.

Exemples

C, Ocaml, Haskell, Java, ...

Exemples

Python, PHP, Javascript, Java, ...

Inconvénient du typage dynamique

- Moins d'erreurs sont capturées à la compilation, plus de bugs peuvent arriver à l'exécution.
- Le typage doit être fait à l'exécution, cela peut rendre le programme moins efficace.

Avantage du typage dynamique

Le typage dynamique facilite l'introspection.

Exemple I

En Ocaml

```
let f x y  = x + z;;
Error: Unbound value z
```

En Python

```
>>> def f(x,y): return (x+z)
```

Et un jour :

```
f(3,5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
```

Exemple I

En Ocaml

```
# let pos l =
      let x = List.filter (fun x -> x > 0) l in
      x+1;;
Error: This expression has type int list
       but an expression was expected of type int
```

Exemple II

En Python

```
>>> def pos(l):
    def f(x):
        return (x>0)
    x= filter(f,l)
    return (x+1)
>>> pos([1,2,3,-5])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in pos
TypeError: can only concatenate list (not "int")
to list
```

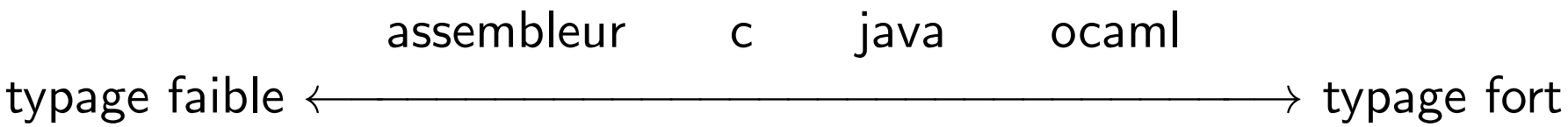
Du typage statique en Python ?

En Python récent, on peut donner des annotations de types.

- Uniquement pour l'IDE
- Pas utilisé par le compilateur

Typage faible/fort

- Pleins de définitions possibles.



Exemple

En Ocaml

```
# let pos l = List.filter (fun x -> x > 0) l;;
val pos : int list -> int list = <fun>
# pos [1;2;-5;-4;-8;-5];;
- : int list = [1; 2]
# pos [1;2.0;-5.5;-4;-8;-5];;
Error: This expression has type float but an expression
  was expected of type int
```

Exemple

En Python

```
>>> def pos(l):
    def f(x):
        return (x>0)
    return (filter (f,l))
>>> pos([1,2,3,-5-2,5])
[1, 2, 3, 5]
>>> pos([1,2.3,3,-5.0-2,5])
[1, 2.2999999999999998, 3, 5]
```

Exemple

En Python

```
def f(x):
  if(2):
    return "toto"
  else:
    return "titi"
```

En Ocaml

```
# let f(x)=
  if (2)
    then "toto"
  else "titi";;
Error: This expression has type int but an expression
    was expected of type bool
```

Exemple

En C

```c
#include <stdio.h>
const unsigned int i = 10;
const int j = -5;
int main(void)
 {
        if (i < j) printf("i plus petit que j\n");
      else printf("i plus grand que j\n");
         return 0;
}
```

Exemple

En C

```
#include <stdio.h>
const unsigned int i = 10;
const int j = -5;
int main(void)
 {
        if (i < j) printf("i plus petit que j\n");
      else printf("i plus grand que j\n");
         return 0;
}
```

```
gcc -Wall -Wextra toto.c
 warning: comparison of integers of different signs: const uns
```

Inférence de type

Définition

Mécanisme qui permet de trouver automatiquement les types associés à des expressions, sans qu'ils soient indiqués explicitement dans le code source.

Exemple

Ocaml, Haskell, Scala, C#3.0,Java > 7 . . ., C++11

Contre-exemple

C, C++03, . . .

Polymorphisme

Définition

- Pouvoir programmer une fonction qui fonctionne pour plusieurs types.
- Décrire le comportement d'une opération de façon générale, i.e. de façon indépendante du type de ses paramètres.
- Permet de réutiliser du code.

Exemples

Python, Ocaml, Haskell, Java > 5 ...

Contre-exemple

C, ...

Plusieurs types de polymorphisme

- Polymorphisme paramétrique
- Polymorphisme ad hoc

Exemple de polymorphisme paramétrique

En Ocaml

```
let rec length l =
match l with
[] -> 0
| a::r -> 1 + length r;;
val length : 'a list -> int = <fun>
```

En Haskell

```
length              :: [a] -> Integer
length []           =  0
length (x:xs)       =  1 + length xs
```

Type somme

En Ocaml

```
type jour = Lundi | Mardi | Mercredi | Jeudi |
Vendredi | Samedi | Dimanche;;
```

Le type somme représente l'union disjointe. Dans certains langages comme (Ocaml, Haskell, Rust, Swift), le compilateur vérifie statiquement que tous les cas de l'union ont été pris en compte par le programmeur.

Hoare's "billion-dollard mistake"

I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years. In recent years, a number of program analysers like PREfix and PREfast in Microsoft have been used to check references, and give warnings if there is a risk they may be non-null. More recent programming languages like Spec# have introduced declarations for non-null references. This is the solution, which I rejected in 1965.

C.A.R. Hoare

`http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mist\ake-Tony-Hoare`

Type option

En Ocaml

```
type 'a option =
  Some of  'a
| None
```

En Python

Optional[X] is equivalent to Union[X, None]

Type produit

Produit cartésien de plusieurs types.

En Python

Tuples :

```
p = (10,22)
```

Tuples avec champs nommés :

```
Point = namedtuple('Point', ['x', 'y'])
p = Point(x=10, y=22)
```

Type produit

En Ocaml

n-uplet

```
type point = float * float
```

n-uplet nommé

```
type point = { abs:float; ord:float}
```

Type de données algébrique

En Python + ADT

```
@adt
class Tree:
    LEAF: Case[int]
    NODE: Case["Tree", "Tree"]
```

Type de données algébrique

En Ocaml

```
type tree=
    Leaf of int | Node of tree *  tree

let exemple = Node(Leaf(2),Leaf(2))
```

Exercice

Proposer de types pour différentes formes d'arbres : arbre binaires, arbre n-aires, expressions arithmétiques, expressions logiques, ...

Fonctions objets de première classe

Exemples

- Scheme (1970s)
- ML family (1970s)
  - Standard ML (1990)
  - Caml (1980s)
  - Ocaml (1996)
  - F# (2010)
- Javascript
- Python (1990)
- C# 3.0 (2007)
- Objective C 2.0 (2007)
- Java SE 7.0 (2010)
- C++11 (2011)

Exemple I

En C# 3.0

```
listOfFoo.Where(x => x.Size > 10);
```

En C++11

```
[](int x, int y) -> int { int z = x + y; return z + x; }
```

En Javascript

```
function bestSellingBooks(threshold) {
  return bookList.filter(
      function (book) { return book.sales >= threshold; }
```

Exemple II

En Ocaml

```
# let f x =
    let g y = x + y in
      g;;
val f : int -> int -> int = <fun>
# let f2=f 5;;
val f2 : int -> int = <fun>
# f2(3);;
- : int = 8
```

En Python

```
>>> def f(x):
    def g(y):
        return(x+y)
    return g
>>> f2=f(5)
>>> f2(3)
8
```

Plan

Curryfication

Définition

La curryfication consiste à transformer une fonction qui prend plusieurs arguments en une fonction qui prend un seul argument et qui retourne une fonction qui prend en arguments les arguments restants.

Remarque : cette opération porte le nom de Haskell Curry (1900-1982).

Exemple

En Caml

Au lieu d'écrire :

```
# let f(x,y) = x + y;;
val f : int * int -> int = <fun>
```

On écrit :

```
# let f x = fun y -> x + y;;
val f : int -> int -> int = <fun>
```

ou bien

```
# let f x y = x + y;;
val f : int -> int -> int = <fun>
```

Pourquoi ?

Pourquoi utiliser la curryfication ?

Pourquoi ?

Pourquoi utiliser la curryfication ? Afin de pouvoir réaliser des *applications partielles* plus facilement.

Exemple application partielle

En Python

```
def mafonction(x,y):
    print(x, "suivi de", y)


titi = functools.partial(mafonction, 'Je suis')
```

Récursivité terminale

Une fonction est dite récursive terminale (*tail-recursive*) quand chaque appel récursif est la dernière instruction exécutée par la fonction.

Non récursive terminale

```
def fact(n):
  if (n==0):
    return 1
  else:
    return (n*fact(n-1))
```

Récursive terminale

```
def fact_tail(n,acc):
  if (n==0):
    return acc
  else:
    return (fact_tail(n-1,n
    *acc))

def fact2(n):
  return fact_tail(n,1)
```

La plupart des compilateurs compilent les fonctions reccursives terminales comme des boucles (inutile d'empiler les appels de fonctions).
Ce n'est pas le cas de Python.

Plan

Gestion de la mémoire

Gestion automatique

Glaneur de cellules, *garbage collector*, ramasse miettes

Exemple

Lisp, Ocaml, Haskell, Python, Java, C#, Perl, PHP ...

Attention !

Un ramasse-miette est toujours une approximation conservatrice de l'objectif idéal de libération des valeurs ne servant plus.

- Consomme plus de mémoire

Gestion manuelle

- Réservation (malloc)
- Libération (free)

Exemple

C, C++, ...

Attention !

- Segfaults.
- Fuites de mémoire.

Gestion de la mémoire

Fuite de mémoire

Il se produit une fuite de mémoire quand un programme ne libère pas la mémoire après l'avoir utilisée.

Erreur de segmentation

Il se produit une erreur de segmentation lorsqu'un programme tente d'accéder à de la mémoire qui ne lui était pas allouée.

Plan

Surcharge

Définition

Plusieurs fonctions peuvent avoir le même nom.

Exemples

- la fonction `print`
- les opérateurs arithmétiques (+ - * /)
- ...

Exemple

En Python

```
>>> 3 + 2.5
```

En Ocaml

```
# 3 + 2.5;;
Error: This expression has type float
but an expression was expected of type int
# 3. +. 2.5;;
- : float = 5.5
```

Surcharge d'opérateurs

En Python

On peut définir une méthode `__add__` pour obtenir la notation infixe.

Plan

Exceptions

Définition

Système de gestion des conditions exceptionnelles pendant l'exécution du programme.
Lorsqu'une exception se produit, l'exécution normale du programme est interrompue et l'exception est traitée.

Exemples

Ada, C++, Java, ML, Objective-C, Ocaml, Prolog, Python, ...

Syntaxe

C++, Java	`try`	`catch`	`throw`
Ocaml	`try`	`with`	`raise`
Python	`try`	`except`	`raise`
Objective-C	`@try`	`@catch`	`@throw`
Ada	`begin`	`exception`	`raise`

Exemple I

En Ocaml

```
#exception Negatif of int;;
exception Negatif of int

#let rec fact n =
  if n<0 then raise (Negatif n) else
   if n=0 then 1 else n*fact(n-1);;
 val fact : int -> int = <fun>
#let afficheFact x =
    try
      print_int(fact(x))
    with
        Negatif x -> print_int x;print_string " est negatif"
      | _         -> print_string " Autre exception";;
val afficheFact : int -> unit = <fun>
```

Exemple II

En Python

```
>>> def fact(n):
    if n<0:
        raise Exception("Negatif")
    elif n==0:
        return 1
    else:
        return (n*fact(n-1))
>>> def afficheFact(x):
    try:
        print (fact(x))
    except Exception as inst:
        print inst
```

Exemple III

En C++

```cpp
#include <string>

int division(int a,int b)
{
    if(b == 0)
        throw std::string("ERREUR : Division par zero !");
    else
        return a/b;
}
```

Plan

Autres aspects pour choisir un langage de programmation

- Portabilité.
- La taille de la bibliothèque standard.
- L'existence de nombreuses bibliothèques.
- L'existence de programmeurs qualifiés.
- La compatibilité avec le code existant : exemple de Fortran en physique, Cobol dans les banques, ...
- La compatibilité avec un outil : exemple de Vba et Excel.
- Existence d'une certification : exemple de Ada en aéronautique.

A propos de l'existence de programmeurs qualifiés

John Carmack fondateur de la société id Software. A propos du typage statique vs dynamique, de l'existence de programmeurs qualifiés...

> *"It's not because people are no good. The very best programmers always make mistakes. This is something that I've really internalized ; that, no matter how good you think you are, you are making mistakes all the time, and you have to have structures around you to try and help you limit the damage that your mistakes will cause, find them as early as possible, so that you can correct them, and the earliest possible time is at compile time. So I'm all about trying to be much more restrictive on what we can do, here. And on the one hand, I would entertain programming in—I'm very tempted to want to move to—a functional programming language ; start programming in Haskell or OCaml or something, but that's not a credible thing to do in the game industry. Because [...], we are back with performance issues, [...] and huge educational issues"*

John Carmack

Plan

Stratégie d'évaluation

Exemple

$$f(x) = 2$$

$$f(10!)?$$

Stratégies d'évaluation

Stratégie stricte L'argument est évalué avant la fonction.
Exemple : La plupart des langages.

Stratégie paresseuse L'argument est évalué seulement si c'est nécessaire.
Exemple : Haskell.

Exemple appel paresseux

Exemple d'une fonction avec appel paresseux dans de nombreux langages :

En Python

```
>>> False and f()
False
>>> def loop():
    loop()
>>> False and loop()
False
>>> def monet(a,b):
   if a:
      return b
   else:
      return False
>>> monet(False,loop())
RuntimeError: maximum
recursion depth exceeded
```

En Ocaml

```
# let rec loop () = loop();;
val loop : unit -> 'a = <fun>
# false & loop();;
- : bool = false
# let monet(a,b) =
   if a then b else false;;
val monet :
  bool * bool -> bool = <fun>
# monet(false,loop())
```

Ordre d'évaluation des arguments

Certains langages définissent l'ordre d'évaluation des arguments (Python G $\rightarrow$ D) d'autres pas (Ocaml).

Attention !

Il vaut mieux ne pas écrire de programmes qui dépendent de l'ordre d'évaluation.

Plan

Appel par valeur et par référence

- Appel par valeur (on passe la valeur de l'objet en la recopiant)
- Appel par référence (on passe une référence à l'objet)
- Appel par adresse
  L'appel par valeur d'une adresse est considéré par abus de langage comme un appel par référence.

Certains langages (Pascal par exemple) permettent de manipuler explicitement les références à un objet.

Exemple I

En Python

```
>>> def tata(x):
...     x=5
...     print x
...     return 3
>>> x=12
>>> tata(x)
5
3
>>> x
12
```

Exemple I

En Python

```
l=[3,2,1,4]
>>> def titi(l):
...     l[0]=5
...     return 3
>>> titi(l)
3
>>> l
[5, 2, 1, 4]
```

Exemple II

En C

```c
int tata(int x){
  x=5;
  printf("%d",x);
  return 3;
}
```

Exemple III

En Ocaml

```
# let f(x) =
  let x = 5 in
  print_int x;
  3;;
val f : 'a -> int = <fun>
# let x = 12;;
val x : int = 12
# f(x);;
5- : int = 3
#x;;
- : int = 12
```

Exemple III

En Ocaml

```
# let f(x) =
   x := 5;
   print_int !x;
   3;;
val f : int ref -> int = <fun>
# let x = ref 12;;
val x : int ref = {contents = 12}
# f(x);;
5- : int = 3
# x;;
- : int ref = {contents = 5}
```

Exemple IV

Exemple en C

```c
void echange (int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

Plan

Portée d'une variable

La portée d'une variable est la zone dans laquelle cette variable est définie.

En C, Java, ...

```
{
   int x=3;
   {
      int x = 2;
      printf("%d", x);
   }
   printf("%d",x);
}
```

Liaison statique/dynamique

Liaison dynamique

La liaison se fait à l'exécution.

Liaison statique :

Pascal, ADA, C, Ocaml, Haskell, Scheme, Perl ...

Liaison dynamique :

Lisp, Python, Perl ...

Exemple I

En Caml

```
# let d = 3;;
val d : int = 3
# let f x = d + x;;
val f : int -> int = <fun>
# f 5;;
- : int = 8
# let d = 4;;
val d : int = 4
# f 5;;
- : int = 8
```

En Python

```
>>> d=3
>>> def f(x):
...     return(x+d)
...
>>> f(5)
8
>>> d=4
>>> f(5)
9
```

Exemple II

En C

```
int x = 0;
int f() { return x; }
int g() { int x = 1; return f(); }
```

Avantage

La liaison dynamique permet d'adapter facilement le fonctionnement d'une fonction à l'état courant du système.

Mais ...

- La liaison dynamique oblige le programmeur à envisager tous les contextes dans lesquels le code peut être appelé.
- Il est plus facile de raisonner sur du code à liaison statique.

Plan

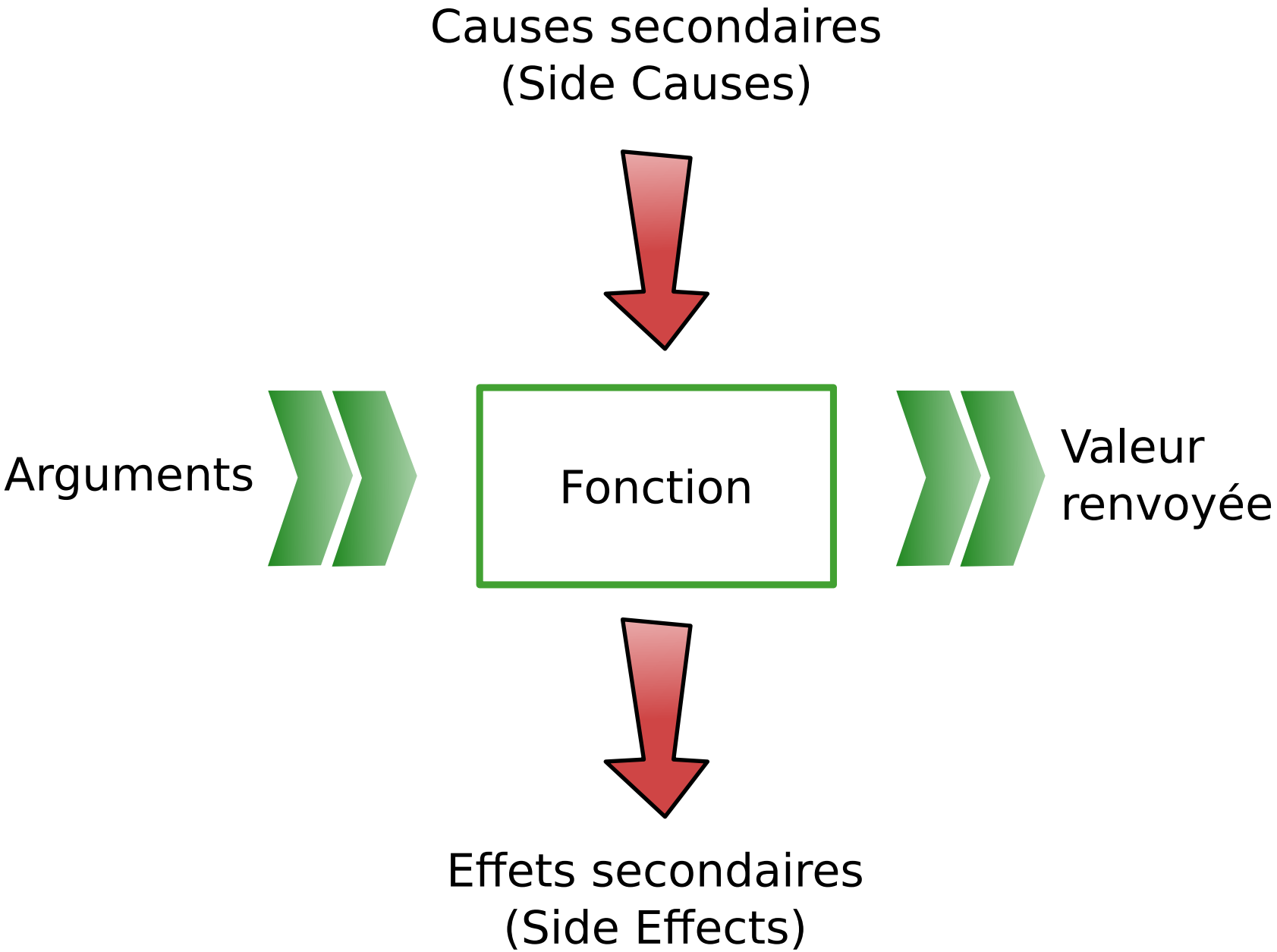
Notion d'effet de bord (*side effect*)

Définition

On dit qu'une fonction réalise un effet de bord quand son exécution ne fait pas que retourner un résultat mais modifie l'état global de l'ordinateur de manière observable.

Exemples d'effets de bord

- modifier une variable statique ou globale
- modifier la valeur pointée par un ou plusieurs de ses arguments
- écrire à l'écran ou dans un fichier


Causes secondaires
(Side Causes)
Arguments
Fonction
Valeur
renvoyée
Effets secondaires
(Side Effects)

Exemple

La fonction suivante modifie l'affichage de l'ordinateur en affichant "Bonjour" puis renvoie la valeur de son argument multipliée par 2.

En C

```c
int mult2(int x) {
    printf("Bonjour\n");
    return(2*x);
}
```

En Python

```python
def mult2(x):
    print("Bonjour\n");
    return(2*x)
```

Conséquence

Cette fonction diffère de la fonction mathématique $x \mapsto 2x$.
En effet, les programmes suivants ne se comportent pas pareil.

En C

```c
int main() {
  int x = 3;
  int y = mult2(x)
              + mult2(x);
  return(1);}

int main() {
  int x = 3;
  int y = 2*mult2(x);
  return(1);}
```

En Python

```python
def main():
    x=3
    y=mult2(x)+mult2(x)
    return(1)


def main():
    x=3
    y=2*mult2(x)
    return(1)
```

L'un écrit `Bonjour` deux fois, l'autre qu'une seule fois.

Deuxième exemple

Considérons les fonctions suivantes :

En C

```c
int n=3;

int f(int x) {
    return(x+n);
}
```

En Python

```python
n=3


def f(x):
    return(x+n)
```

Elles ne réalisent pas d'effets de bord, mais ne se comportent pas comme des fonctions au sens mathématique.

Limiter les effets de bords en Python

x=5

```
def safedouble(x)
    x = 2 * x
    return x

def unsafedouble(x)
    global x
    x = 2 * x
    return x
```

Fonction pure

Définition

Une fonction est pure si elle ne dépend que de ses arguments.

Exemples de fonctions impures :

- une fonction qui lit une donnée au clavier
- une fonction dont le comportement dépend d'une variable globale
- une fonction qui renvoie l'heure

Troisième exemple

Exemple d'une fonction impure réalisant un effet de bord :

En C

```
int n = 2;
int inc(int k) { n = n + k; return n; }
f(inc(1) + inc(1));
```

Il est impossible de remplacer `inc(i) + inc(i)` par `2 * inc(i)` car la valeur de `inc(i)` diffère à chaque appel.

Ce comportement est fondamentalement différent de celui d'une fonction mathématique.

Transparence référentielle

Définition

Une expression est référentiellement transparente si elle peut être remplacée par sa valeur.

Attention !

Les effets de bord compliquent grandement la compréhension des programmes et sont la source de bogues.

Mais !

On ne peut pas écrire de programme intéressant qui ne réalise **aucun** effet de bord. En effet, tout programme devra à un moment ou un autre interagir avec l'utilisateur, afficher une information à l'écran, imprimer un document, écrire un fichier sur le disque dur...

Le paradigme de la programmation fonctionnelle consiste à écrire le moins possible de fonctions qui réalisent des effets de bord. C'est donc un style de programmation qui est à encourager dans tous les langages.

Résumé

Attention !

Les fonctions qui réalisent des effets de bords ne correspondent pas au concept mathématique de fonction.

Les fonctions *pures* et *sans effet de bord* correspondent aux fonctions (calculables) au sens mathématique.

Les effets de bords, c'est mal ! mais on ne peut pas s'en passer.

Fonctions vs procédures

On appelle souvent une fonction qui ne renvoie rien une procédure.
Une procédure qui ne réalise pas d'effet de bord ne sert à rien.
Le type de retour des procédures est `void` en C et `unit` en Caml `NoneType` en Python.

Il faut savoir distinguer une *action* et une *expression*.

Exemple

En Ocaml

```
# let f(x) =
   print_string("x vaut: ");
   print_int(x);;
val f : int -> unit = <fun>
# let g(x) =
    let y = x + 2 in
      ();;
Warning Y: unused variable y.
val g : int -> unit = <fun>
# let h(x) =
    let y = x + 2  in
      y
val h : int -> int = <fun>
```

En Python

```
>>> def f(x):
   print("x vaut: ")
   print(x)
>>> def g(x):
      y=x+2
>>> g(2)
>>> def h(x):
    y=x+2
    return y
>>> h(3)
5
>>> z=g(2)
>>> type(z)
<type 'NoneType'>
```

Affectation vs égalité

Affectation

Modifie la valeur d'une case mémoire.
C'est une *action*.

Egalité

Teste si deux valeurs sont égales.
C'est une fonction qui renvoie un *booléen*.

Syntaxe

	Affectation	Egalité
C	=	==
Ocaml	:=	=
Python	=	==

Pointeurs

Définition d'un pointeur

Un pointeur est une variable qui contient une adresse mémoire et non une valeur. Cette adresse mémoire est en général l'adresse d'une autre variable : on dit que le pointeur pointe sur la variable.

Opérations de bases sur les pointeurs

Déréférencer

Déréférencer un pointeur consiste à extraire la valeur de la variable sur laquelle il pointe.

Dangling pointeur

Les pointeurs pendants sont ceux qui n'ont pas été initialisés ou bien qui pointent sur une zone mémoire qui a été désallouée.

```
int * toto(void)
{
    int i = 42;
    return &i;
}
```

Solution partielle

Mettre le pointeur à NULL après désallocation.

Mais...

il peut y avoir plusieurs pointeurs qui pointent sur la même zone.

Pointeur vs référence

On peut distinguer deux manières de faire référence à un objet :

Définition d'un pointeur

Un pointeur est une variable qui contient une adresse mémoire et non une valeur. Cette adresse mémoire est en général l'adresse d'une autre variable : on dit que le pointeur pointe sur la variable.

Définition d'une référence

Une référence est un 'alias' pour une autre variable. La référence doit être initialisée vers une certaine variable, mais ne pourra par la suite plus se référer à une autre variable.

La différence est qu'un pointeur peut être déplacé, i.e. qu'on peut lui changer l'adresse vers laquelle il pointe. Alors que la référence ne peut pas se référer à une autre variable si elle est déjà référencée sur une certaine variable.

Exemple

En Ocaml

```
# let x = ref 2;;
val x : int ref = {contents = 2}
# x := !x +1;;
- : unit = ()
# !x;;
- : int = 3
```

Résumé

Attention !

Les pointeurs sont à manipuler avec précaution.

Mais !

On ne peut pas s'en passer.

Histoire des langages de programmation I

1951 Regional Assembly Language

1952 AUTOCODE

1954 FORTRAN

1958 LISP

1958 ALGOL 58

1959 COBOL

1962 Simula I (Simple universal language)

Histoire des langages de programmation II

1970 Pascal

1970 Forth

1972 C

1972 Smalltalk

1972 Prolog

1973 ML

1978 SQL

Histoire des langages de programmation III

1983 Ada

1983 C++

1985 Eiffel

1987 Perl

Histoire des langages de programmation IV

1990 Haskell

1991 Python

1991 Java

1993 Ruby

1993 Lua

1994 ANSI Common Lisp

1995 JavaScript

1995 PHP

2000 C#