Programmation fonctionnelle

Programmation en Caml

Samuele Giraudo

samuele.giraudo@univ-eiffel.fr
https://igm.univ-mlv.fr/~giraudo

Université Gustave Eiffel
LIGM, bureau 4B162

2021–2022

Prélude n°1

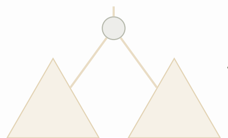
Les arbres binaires de recherche en programmation fonctionnelle

Arbres binaires

Un arbre binaire est

1. soit une feuille

2. soit un nœud attaché à deux arbres binaires



C'est une définition **récursive** car la définition de la notion fait référence à la notion elle-même.

Les arbres binaires sont en général très faciles à implanter dans des langages fonctionnels.

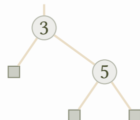
Définition du type arbre binaire en CAML :

```
type arbre_b =
    |Feuille
    |Noeud of arbre_b * int * arbre_b
```

Construction d'un arbre binaire en CAML :

```
Noeud (Feuille, 3, Noeud (Feuille, 5, Feuille))
```

Cette expression désigne l'arbre binaire



Insertion dans un arbre binaire de recherche

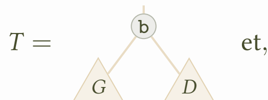
Soit T arbre binaire de recherche.

On note $T_{\leftarrow \mathtt{a}}$ l'arbre binaire obtenu après **insertion** de la lettre $\mathtt{a}$ dans T.

Cet arbre se définit de la manière suivante :

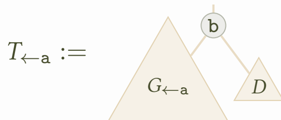
1. si $T = \square$, alors

2. sinon, T est de la forme

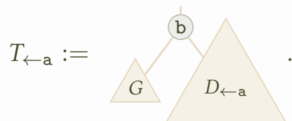


et,

2.1 si $\mathtt{a} \leqslant \mathtt{b}$, alors



2.2 sinon $\mathtt{a} > \mathtt{b}$ et



.

Insertion dans un arbre binaire de recherche en Caml

Définition de la fonction (récursive) d'insertion dans un arbre binaire de recherche en Caml :

```
let rec inserer t a =
    match t with
        |Feuille -> Noeud (Feuille, a, Feuille)
        |Noeud (g, b, d) when a <= b -> Noeud (inserer g a, b, d)
        |Noeud (g, b, d) -> Noeud (g, b, inserer d a)
```

Construction d'un arbre binaire de recherche par insertions successives :

```
let t1 = inserer Feuille 3 in
let t2 = inserer t1 4 in
let t3 = inserer t2 2 in
inserer t3 1
```

Instructions *vs* expressions

Calcul d'une sous-liste

Considérons le problème qui, étant donnée une liste `lst` en entrée, consiste à calculer la sous-liste de `lst` obtenue en prenant un élément sur deux.

Sous le paradigme impératif, on propose en PYTHON la fonction ci-contre :

```
def f(lst) :
    res = []
    for i in range(len(lst)) :
        if i % 2 == 0 :
            res.append(lst[i])
    return res
```

Le calcul de `f([0, 1, 2, 3, 4])` utilise de la mémoire :

- pour construire le résultat `res` ;
- pour maintenir la variable `i` ;
- éventuellement pour les fonctions appelées (`range`, `append`, *etc.*) ;
- pour enregistrer l'état de la machine (adresse de l'instruction exécutée).

Les variables `res` et `i` sont lues et modifiées au cours du temps.

Calcul d'une sous-liste en fonctionnel

Sous le paradigme fonctionnel, on aurait plutôt (en CAML pratiqué par un débutant) :

```
let rec f lst =
    if (List.length lst) <= 1 then
        lst
    else
        (List.hd lst) :: (f (List.tl (List.tl lst)))
```

Pour calculer l'**expression** `f([0; 1; 2; 3; 4])`, on l'**évalue** :

`f([0; 1; 2; 3; 4])` $\rightarrow$ `0 :: f([2; 3; 4])` $\rightarrow$ `0 :: 2 :: f([4])` $\rightarrow$ `0 :: 2 :: [4]` $\rightsquigarrow$ `[0; 2; 4]`

Dans ce cas, il n'y a

- ni utilisation externe de la mémoire ;
- ni état d'avancement du calcul qui dépend du temps.

Le calcul se fait en **réécrivant** l'expression tant que possible. La définition de **fonctions** permet d'expliquer comment réaliser les réécritures.

Principes généraux

Quelques caractéristiques du CAML

- **Allocation** de mémoire **implicite**.

  $\rightsquigarrow$ le programmeur se concentre sur des aspects moins techniques et plus importants.

- Programmer = **penser**.

  $\rightsquigarrow$ l'écriture d'un programme est rapide et est fidèle aux définitions des objets et algorithmes manipulés.

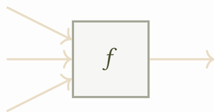
- Le **compilateur** est **exigeant** et vérifie beaucoup de choses.

  $\rightsquigarrow$ en pratique :

« Le compilateur CAML *n'accepte presque jamais mon code mais quand il l'accepte, ça marche comme je le voulais. »*	*vs*	*« Le compilateur* X *accepte presque toujours mon code mais ça ne marche presque jamais comme je le voulais. »*

Principes de la programmation fonctionnelle

La programmation fonctionnelle est un paradigme de programmation dans lequel les objets de base sont les **expressions** et les **fonctions** :



On **évite les effets secondaires** (et donc, on ne fait pas d'affectation).

On **évite les séquences d'instructions impératives** (et donc, on n'utilise pas de boucles `while`, `do while`, `for` ou autres).

On utilise en revanche l'**application de fonctions** et la **récursivité**.

Pré-requis

Ce cours demande les pré-réquis suivants :

- des connaissances avancées en **programmation générale** (écriture de programmes, gestion de projets) ;
- des connaissances avancées en **programmation impérative** (instructions, variables, fonctions, effets secondaires) ;
- des connaissances solides du **langage** C ou du **langage** PYTHON ;
- des connaissances de base en **algorithmique** (récursivité, manipulation de listes, d'arbres).

Objectifs du cours

Ce module de programmation fonctionnelle a pour buts

1. d'avoir une 1re approche du **paradigme de programmation fonctionnelle** ;
2. d'apprendre les bases de **programmation en Caml** en mode fonctionnel.

Celui-ci est organisé en trois axes.

1. **Axe 1** : **bases théoriques**.

   Machines de Turing, décidabilité et indécidabilité, paradigmes de programmation, caractéristiques des langages de programmation.

2. **Axe 2** : **concepts premiers**.

   Programmation en Caml, liaisons, fonctions, entrées / sorties, compilation, types.

3. **Axe 3** : **concepts avancés**.

   Récursivité terminale, filtrage, fonctions d'ordre supérieur, polymorphisme, stratégies d'évaluation, opérations sur les listes, non mutabilité, λ-calcul.

Contenu du cours

Axe 1.

1. Théorie

2. Caractéristiques

Axe 2.

1. Programmation

4. Pratique

5. Types

Axe 3.

6. Notions

7. Listes

8. λ-calcul

Bibliographie

Bibliographie non exhaustive :

- X. Leroy, P. Weis, *Le langage Caml*, Dunod, 2e édition, 2009.
  Lien : `http://caml.inria.fr/distrib/books/llc.pdf`
- E. Chailloux, P. Manoury, B. Pagano, *Développement d'applications avec Objective Caml*, O'Reilly, 2000.
- X. Leroy *et al.*, The OCaml system release 4.13, September 24, 2021.
  Lien : `http://caml.inria.fr/pub/docs/manual-ocaml/`
- G. Dowek, J.-J. Lévy, *Introduction à la théorie des langages de programmation*, École Polytechnique, 2006.
- C. Okasaki, *Purely Functional Data Structures*, Cambridge University Press, 1999.
- S. L. Peyton Jones, D. Lester, *Implementing Functional Languages*, Prentice Hall, 1992.
- P. Hudak, D. Quick, *The Haskell School of Music : From Signals to Symphonies*, Cambridge University Press, 2018.

Axe 1 : bases théoriques

1. Théorie

2. Caractéristiques

1. Théorie

Plan

1. Théorie

1.1 Machines de Turing
1.2 Décidabilité et indécidabilité
1.3 λ-calcul

Brève chronologie de la programmation

- 1801 : 1re machine programmable : le **métier à tisser** de Jackard.
- 1920 : M. Schönfinkel introduit la **logique combinatoire**.
- 1936 : Church introduit le **λ-calcul**.
- 1936 : Turing invente la **machine de Turing**.
- Années 1940 : programmation en assembleur et en langage machine.
- Années 1950-1960 : 1ers vrais **langages de programmation** : FORTRAN, COBOL et Lisp.
- Années 1960-1970 : **paradigmes de programmation** : impératif, fonctionnel, orienté objet, logique.

Machines de Turing

Machine de Turing : machine théorique qui fournit une abstraction des appareils de calcul.

Thèse de Church-Turing :

« tout ce qui est effectivement *calculable*
est calculable par une machine de Turing ».

Une machine de Turing est un quadruplet (E, i, t, Δ) où

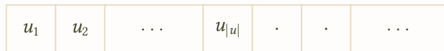
1. E est un ensemble fini d'états ;
2. $i \in E$ est l'état initial ;
3. $t \in E$ est l'état terminal ;
4. $\Delta : E \times \{\cdot, 0, 1\} \to E \times \{\cdot, 0, 1\} \times \{\mathrm{G}, \mathrm{D}\}$ est une fonction de transition.

Fonctionnement des machines de Turing

Soit $M := (E, i, t, \Delta)$ une machine de Turing et $u \in \{0, 1\}^*$ un mot.

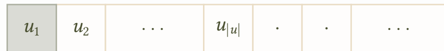
L'exécution de M sur u consiste à :

1. placer u le plus à gauche dans un tableau infini à droite, le ruban :



   Les cases à droite de u sont remplies jusqu'à l'infini de $\cdot$.

2. Placer la tête de lecture / écriture sur la 1re case du ruban :



   On appelle a la lettre dans $\{\cdot, 0, 1\}$ indiquée par la tête de lecture / écriture à un instant donné.

3. Affecter au registre d'état e la valeur i (l'état initial).

4. Réaliser les actions dictées par Δ, le programme.

Fonctionnement des machines de Turing

Pour réaliser les actions dictées par le programme de M, on procède séquentiellement comme suit :

(1) calculer $(e', a', s') := \Delta(e, a)$.

(2) Écrire a' dans la case du ruban indiquée par la tête de lecture / écriture.

(3) Affecter au registre d'état e l'état e'.

(4) Si $s' = \mathrm{D}$, déplacer la tête de lecture / écriture d'un pas vers la droite ; sinon, la déplacer si possible d'un pas vers la gauche.

(5) Si $e = t$, alors l'exécution est terminée ; sinon, revenir en (1).

Le résultat de l'exécution de M sur u est le mot $M(u)$ défini comme étant plus court préfixe du ruban qui contient tous ses 0 et ses 1.

Exemple : complémentaire d'un mot

– Exemple –

Soit $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ la machine de Turing dont le programme Δ est défini par

$$(e_1, 0) \mapsto (e_1, 1, \mathrm{D}),$$
$$(e_1, 1) \mapsto (e_1, 0, \mathrm{D}),$$
$$(e_1, \cdot) \mapsto (e_2, \cdot, \mathrm{D}).$$

Registre d'état	Ruban
e_1	**[0]** 0 1 · · ...
e_1	1 **[0]** 1 · · ...
e_1	1 1 **[1]** · · ...
e_1	1 1 0 **[·]** · ...
e_2	1 1 0 · **[·]** ...

Voici les étapes du calcul de $M(001)$. L'exécution de M sur u fournit ainsi le résultat $M(001) = 110$.

Ce programme Δ permet de calculer le complémentaire de tout mot $u \in \{0,1\}^*$ en entrée.

Exécutions qui ne se terminent pas

Il est possible de construire des machines de Turing telles que pour certaines entrées u, l'exécution de M sur u **ne se termine pas**.

– Exemple –

Soit la machine de Turing $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ dont le programme Δ est défini par

$$(e_1, 0) \mapsto (e_1, 0, \mathrm{D}), \quad (e_1, 1) \mapsto (e_2, 1, \mathrm{G}), \quad (e_1, \cdot) \mapsto (e_1, \cdot, \mathrm{D}).$$

Calcul de $M(001)$:

Registre d'état	Ruban
e_1	[0] 0 1 · · …
e_1	0 [0] 1 · · …
e_1	0 0 [1] · · …
e_2	0 [0] 1 · · …

On arrive à e_2 qui est l'état terminal, l'exécution est terminée.

Calcul de $M(000)$:

Registre d'état	Ruban
e_1	[0] 0 0 · · …
e_1	0 [0] 0 · · …
e_1	0 0 [0] · · …
e_1	0 0 0 [·] · …
e_1	0 0 0 · [·] …

La tête de lecture / écriture part vers infini, l'exécution ne se termine pas.

Coder une machine de Turing par un entier naturel

On associe à toute **machine de Turing** M un **entier naturel** $\mathrm{code}(M)$ de la manière suivante :

1. on fixe des conventions (arbitraires mais rigoureuses) pour écrire les définitions des machines de Turing avec des caractères ASCII ;
2. on considère la suite de caractères m ainsi obtenue qui code M ;
3. on considère la suite de bits u obtenue en remplaçant chaque caractère de m par sa représentation binaire ;
4. on obtient finalement l'entier naturel $\mathrm{code}(M)$ en considérant l'entier dont u est la représentation binaire.

Coder une machine de Turing par un entier naturel

– Exemple –

Considérons la machine de Turing $M := (\{e_1, e_2\}, e_1, e_2, \Delta)$ dont le programme Δ est défini par

$$(e_1, 0) \mapsto (e_1, 0, \mathrm{D}), \quad (e_1, 1) \mapsto (e_2, 1, \mathrm{G}), \quad (e_1, \cdot) \mapsto (e_1, \cdot, \mathrm{D}).$$

M se code en caractères ASCII en le texte m suivant :

```
etats : e1, e2
initial : e1
terminal : e2
```

```
delta : (e1, 0) -> (e1, 0, D)
delta : (e1, 1) -> (e2, 1, G)
delta : (e1, .) -> (e1, ., D)
```

On en déduit la représentation binaire

$$u = \texttt{01100101}\ \texttt{01110100} \cdots \texttt{00101001}$$

et de celle-ci, l'entier naturel $\mathrm{code}(M)$. En décimal, ce nombre est

$\mathrm{code}(M) =$ 1195273483522463947698188428566573994862939056290801762903598135757140294873881005500260610437207957403372
269791891457737724736737165168419101329132422751418637824051118841640585439990747361814064299553649044252
299751665450294668928324126341232682416673453539993886044581503368944491857205586247218648808828981756969 .

Programmes et entiers

Ainsi, lorsque des conventions de codage ont été spécifiées, l'application code est injective (deux machines de Turing différentes ne peuvent pas avoir un même code).

On peut ainsi ordonner l'ensemble de toutes les machines de Turing : on pose $M \leqslant M'$ si $\mathrm{code}(M) \leqslant \mathrm{code}(M')$.

Les machines de Turing peuvent donc être ordonnées en un segment infini

$$M_0 < M_1 < M_2 < \cdots < M_n < \cdots$$

Le rang $\mathrm{rang}(M)$ d'une machine de Turing M est la position de M dans ce segment.

L'application rang fournit une bijection entre l'ensemble des machines de Turing et l'ensemble des entiers naturels.

En conclusion un **programme** (une machine de Turing) est un **entier** et réciproquement.

Plan

1. Théorie

1.1 Machines de Turing

1.2 Décidabilité et indécidabilité

1.3 λ-calcul

Problèmes de décision

Un problème de décision est une question P qui prend un mot de $\{0, 1\}^*$ en entrée et qui répond « oui » ou « non ».

– Exemples –

- P : le mot est-il un palindrome ? $P(010010) = \text{oui}$, $P(011) = \text{non}$;
- P : la longueur du mot est-elle paire ? $P(\epsilon) = \text{oui}$, $P(1) = \text{non}$;
- P : l'entier en base deux codé par le mot est-il premier ? $P(111) = \text{oui}$, $P(100) = \text{non}$;
- P : le mot est-il le codage binaire d'un programme C accepté à la compilation par `gcc` avec l'option `-ansi` ?

Décidabilité et indécidabilité

Un problème de décision P est décidable s'il existe une machine de Turing M_P telle que pour toute entrée $u \in \{0,1\}^*$, l'exécution de M_P sur u se termine et

$$M_P(u) = \begin{cases} 1 & \text{si } P(u) = \text{oui}, \\ 0 & \text{sinon.} \end{cases}$$

Lorsqu'il n'existe pas de telle machine de Turing, P est dit indécidable.

Intuitivement, un problème de décision P est décidable s'il est possible d'écrire, dans un langage suffisamment complet, une fonction f paramétrée par un objet u renvoyant `true` si $P(u) = \text{oui}$ et `false` sinon.

Le problème de l'arrêt

Le problème de l'arrêt est le problème de décision Arr prenant en entrée le codage binaire u d'un programme f et renvoyant oui si l'exécution du programme f se termine et non sinon.

Le problème de l'arrêt est **indécidable**.

Intuitivement, cela dit qu'il est impossible de concevoir un programme qui accepte en entrée un autre programme f et qui teste si l'exécution de f se termine.

Indécidabilité du problème de l'arrêt

On montre que Arr est indécidable par l'absurde en supposant que Arr est décidable.

Il existe donc une machine de Turing M_{Arr}.

On se base sur cette existence pour construire l'entier positif absurde défini par les instructions :

(1) soit E l'ensemble vide ;

(2) pour toute suite d'instructions f qui s'exprime avec moins ou autant de caractères que ces instructions définissant absurde,

   (a) si $M_{\text{Arr}}(u) = 1$, où u est le codage binaire de f :

      (i) ajouter à E la valeur calculée par f, exprimée par un entier.

(3) Renvoyer $\min(\mathbb{N} \setminus E)$.

Intuitivement, absurde est le plus petit entier qu'il est impossible de définir par une suite d'instructions ayant moins ou autant de caractères que absurde.

On vérifie facilement que l'exécution de absurde se termine. Ainsi, la valeur de retour d'absurde figure dans E (étape (a)).

Par ailleurs, la valeur renvoyée par absurde ne figure pas dans E (étape (3)). Ceci est absurde : M_{Arr} n'existe pas et Arr est donc indécidable.

Plan

1. Théorie

1.1 Machines de Turing
1.2 Décidabilité et indécidabilité
1.3 λ-calcul

Fonctions récursives

Une fonction récursive est une fonction

$$f : \mathbb{N}^k \to \mathbb{N}, \qquad k \geqslant 0,$$

qui est intuitivement calculable.

– Exemples –

- $f : (n_1, n_2, n_3) \mapsto n_1 + n_2 + n_3$;
- $f : n \mapsto 1$ si n est pair, 0 sinon ;
- $f : n \mapsto 1$ si $n \leqslant 1$, $n \times f(n-1)$ sinon ;

sont des fonctions récursives.

– Exemple –

En revanche, la fonction $f : \mathbb{N} \to \mathbb{N}$ définie par

$$f(n) := \begin{cases} 1 & \text{si } \mathrm{Arr}(g) = \text{oui où } g \text{ est le prog. tq. } \mathrm{rang}(g) = n, \\ 0 & \text{sinon,} \end{cases}$$

n'est pas une fonction récursive (si elle était calculable, le problème de l'arrêt serait décidable).

λ-calcul

Le λ-calcul a été introduit dans le but de proposer un formalisme pour définir et décrire les fonctions récursives.

En λ-calcul, la notion première est celle d'**expression**. Les **fonctions** sont des expressions particulières.

Une expression du λ-calcul est

1. soit une variable, notée x, y, z, $\ldots$;
2. soit l'application d'une expression f à une expression g, notée $f\ g$;
3. soit l'abstraction d'une expression f, notée $\lambda x.f$ où x est une variable.

– Exemple –

$(\lambda z.z)((\lambda x.x)(\lambda y.((\lambda x.x)y)))$ est une expression.

La β-substitution est le mécanisme qui permet de simplifier (calculer) une expression. Il consiste, étant donnée une expression de la forme $(\lambda x.f)g$ à la simplifier en substituant g aux occurrences libres de x dans f.

2. Caractéristiques

Plan

2. Caractéristiques

2.1 Impératif *vs* fonctionnel

2.2 Caractéristiques des langages

Paradigmes de programmation

Un paradigme est une manière d'observer le monde et d'interpréter la réalité. Il influe sur la manière de penser et d'agir.

Un paradigme de programmation conceptualise la manière de représenter les objets informatiques et de formuler les algorithmes qui les manipulent.

Il existe deux principaux paradigmes de programmation :

1. le paradigme **impératif** ;
2. le paradigme **fonctionnel**.

Le 1er se base sur la machine de Turing, le 2e sur le λ-calcul.

Le paradigme impératif

En programmation impérative, un problème est résolu en décrivant étape par étape les actions à réaliser.

Les éléments suivants sont constitutifs de ce paradigme :

1. les instructions d'affectation ;
2. les instructions de branchement ;
3. les instruction de boucle ;
4. les structures de données mutables.

Des instructions peuvent **modifier l'état de la machine** en altérant sa mémoire.

Le paradigme fonctionnel

En programmation fonctionnelle, la notion de **fonction** est centrale.

Un programme est un emboîtement de fonctions (dans une fonction principale).

L'exécution d'un programme est l'évaluation sur des arguments données de la fonction principale.

Les éléments suivants sont constitutifs de ce paradigme :

1. la liaison d'un nom à une valeur ;
2. la récursivité ;
3. les instructions de branchement ;
4. les structures de données non mutables.

Il n'y a **pas de notion d'état de la machine** car celui-ci ne peut pas être modifié.

Transparence référentielle

Le principe de transparence référentielle stipule que dans tout programme, il est possible de remplacer une expression par sa valeur sans que cela ne change le résultat.

– Exemple –

Considérons le code C suivant :

```
int f(int n) {
    printf("a");
    return n;
}

int g(int n) {
    return n;
}
...
g(f(1));
```

L'expression `f(1)` a pour valeur `1` mais `g(1)` et `g(f(1))` ne produisent pas le même résultat.

En effet, `g(f(1))` affiche `a` mais pas `g(1)`.

Le principe de transparence référentielle n'est donc pas respecté en C.

Règle : en programmation fonctionnelle, le principe de transparence référentielle s'applique.

Ce que nous ferons

Dans la suite de ce module, on adoptera au maximum le paradigme fonctionnel.

Ainsi, nous nous interdirons

1. d'utiliser des variables (et donc aussi des affectations) ;
2. d'utiliser des instructions de boucle ;
3. de produire des effets de bord (sauf éventuellement pour la gestion des entrées/sorties).

En revanche, nous utiliserons de manière courante

1. les **fonctions récursives** ;
2. les **fonctions locales**.

Une variable peut-être vue comme une **fonction d'arité zéro** (c.-à-d. une fonction qui ne prend pas d'entrée).

Plan

2. Caractéristiques

2.1 Impératif *vs* fonctionnel

2.2 Caractéristiques des langages

Vérification de types dynamique *vs* statique

Dans la plupart des langages de programmation, lors d'un appel à une fonction ou de l'évaluation d'une expression, les valeurs mises en jeu doivent être d'un type « autorisé ».

Il y a deux stratégies pour **vérifier les types** (typer) :

1. vérification dynamique, où les types sont vérifiés lors de l'**exécution** du programme ;
2. vérification statique, où les types sont vérifiés lors de la **compilation** du programme.

Vérification de types dynamique

– Exemple –

Considérons le programme PYTHON

```
n = int(input())
if n % 2 == 0 :
    print(n / 2)
else :
    print(n[1])
```

Lors de son **exécution**, des erreurs de typage peuvent se produire :

- si l'utilisateur saisit un entier, l'expression `int(input())` en l. 1 est bien typée ; sinon, elle ne l'est pas ;
- si l'utilisateur saisit un entier pair, l'expression `n / 2` en l. 3 est bien typée ; sinon, c'est l'expression `n[1]` qui est évaluée. Celle-ci n'est pas bien typée car `n` n'est pas un tableau.

  Cette information n'est donnée que lors de l'exécution :

```
Traceback (most recent call last):
    File "Prog.py", line 5, in <module>
        print(n[1])
TypeError: 'int' object has no attribute '__getitem__'
```

Vérification de types statique

– Exemple –

Considérons le programme C

```
#include <stdio.h>
int main() {
    int n;
    scanf("%d", &n);
    if (n % 2 == 0)
        printf("%d\n", n / 2);
    else
        printf("%d\n", n[1]);
    return 0;
}
```

Lors de sa **compilation**, une erreur de typage se produit : `n` est une variable de type `int` mais elle est traitée en l. 8 comme une variable de type `int *`.

Cette information est donnée lors de la compilation :

```
Prog.c: In function ‘main’:
Prog.c:8:25: error: subscripted value is neither array nor pointer nor vector
        printf("%d\n", n[1]);
                        ^
```

Avantages et inconvénients

Vérification de types dynamique.

- Avantage : grande flexibilité dans l'écriture des programmes.
- Inconvénients : les problèmes de typage ne sont mis en évidence qu'à l'exécution (il faut faire attention à tester tous les cas de figure), perte d'efficacité lors de l'exécution (à cause du typage).

Vérification de types statique.

- Avantages : sécurité lors de l'écriture du programme (les erreurs les plus courantes sont détectées à la compilation), bonne efficacité lors de l'exécution.
- Inconvénient : moins de flexibilité dans l'écriture des programmes.

Attribution de types explicite *vs* implicite

La plupart des langages de programmation manipulent des variables et des valeurs qui possèdent un type (ou même plusieurs).

Il y a deux manières principales de **connaître les types** des variables et fonctions utilisées dans un programme :

1. attribution explicite (nommé parfois *Church-style*), où les **types** des variables et fonctions sont **mentionnés** dans le programme ;
2. attribution implicite (nommé parfois *Curry-style*), où les **types** des variables et fonctions ne sont **pas mentionnés** dans le programme.

   Ils sont devinés lors de la compilation (si vérification statique) ou lors de l'exécution (si vérification dynamique) en fonction du contexte.

   Ce mécanisme s'appelle l'inférence des types.

Attribution explicite

– Exemple –

Considérons le programme C

```
#include <stdio.h>

typedef struct {
    int x, y;
} Point2D;

typedef struct {
    int x, y, z;
} Point3D;

void afficher(Point2D *p) {
    printf("(%d, %d)", p->x, p->y);
}

int main() {
    Point3D p;
    p.x = 1, p.y = 2, p.z = 3;
    afficher(&p);
    return 0;
}
```

Dans la fonction `main`, on a demandé explicitement que la variable `p` soit de type `Point3D`. La fonction `afficher` demande l'existence des champs `x` et `y` de son argument. Ces champs existent dans `p` et cependant, ce code ne compile pas.

En effet, même si une variable de type `Point3D` semble pouvoir être utilisée comme une variable de type `Point2D`, ceci est impossible en C qui est un langage à attribution de types explicite : le type de `p` a été fixé lors de la déclaration de `afficher`.

Attribution implicite

– Exemple –

Considérons le programme Caml

```
let suivant x = x + 1 in
print_int (suivant 5)
```

Ici, `suivant` est une fonction. Le type de son paramètre `x` n'est pas spécifié.

Cependant, le fait que l'on réalise une opération arithmétique (`x + 1`) avec ce paramètre indique qu'il s'agit d'un entier.

De cette manière, le mécanisme d'inférence des types suggère au compilateur que `suivant` est une fonction qui accepte un entier et qui renvoie un entier.

Cette valeur renvoyée peut donc être traitée par la fonction `print_int` qui accepte en entrée des valeurs entières.

Avantages et inconvénients

Attribution explicite.

- Avantages : meilleure lisibilité des programmes. En général, la compilation et l'exécution sont plus efficaces.
- Inconvénients : programmes verbeux.

Attribution implicite.

- Avantages : programmes plus concis, le programmeur ne se préoccupe pas d'indiquer les types : le compilateur se charge de trouver ceux qui sont les plus adaptés.
- Inconvénients : programmes moins lisibles. Si la vérification de types est statique, la compilation peut être plus longue (il faut deviner les types). Si la vérification de types est dynamique, l'exécution peut être moins efficace.

Portée statique *vs* dynamique

La portée d'un identificateur dans un programme désigne l'endroit dans lequel il est possible de l'utiliser.

Les identificateurs dans un langage de programmation peuvent être de deux principales catégories :

1. de portée statique, où chaque occurrence d'un identificateur doit être mis en relation avec sa définition préalable ;
2. de portée dynamique, où ce que représente un identificateur peut dépendre de l'exécution du programme (dans certains *mauvais* cas, il peut même ne rien représenter du tout).

Portée statique

– Exemple –

Considérons le programme Caml

```
let fact n =
    if n <= 1 then
        1
    else
        n * (fact (n - 1))
in
fact 3
```

Lors de sa **compilation**, une erreur se produit : l'identificateur `fact`, utilisé en l. 5 est encore non défini.

On obtient le message suivant du compilateur :

```
File "Prog.ml", line 5, characters 12-16:
Error: Unbound value fact
```

Portée dynamique

– Exemple –

Considérons le programme Python

```
n = int(input())
if n == 0 :
    res = "a"
elif n == 1 :
    res = [1, 2, 3]
print(res)
```

Lors de son exécution, l'identificateur `res` en l. 6 peut faire référence à des définitions différentes :

- si l'utilisateur saisit `0`, `res` est défini comme étant la chaîne `"a"` ;
- si l'utilisateur saisit `1`, `res` est défini comme étant la liste `[1, 2, 3]` ;
- dans tous les autres cas, `res` est un identificateur non défini. Cette information est donnée, lors de l'**exécution**, par

```
Traceback (most recent call last):
    File "Prog.py", line 6, in <module>
        print(res)
NameError: name 'res' is not defined
```

Langages fonctionnels purs *vs* impurs

Il y a deux sortes de langages fonctionnels :

1. les langages fonctionnels purs, où tout effet de bord est interdit. Des problèmes se posent notamment lors de la gestion des entrées sorties (mais qui sont résolus élégamment au moyen des *monades*).

2. les langages fonctionnels impurs, où certaines particularités des langages impératifs sont utilisables, comme la gestion classique des entrées / sorties, les affectations ou encore les instructions de boucle.

Caractéristiques des principaux langages

Langage	Vérif. dyn.	Vérif. stat.	Attr. expl.	Attr. impl.	Imp.	Fonc.
C	Non	Oui	Oui	Non	Oui	Non
Python	Oui	Non	Non	Oui	Oui	Impur
Caml	Non	Oui	*Non*	Oui	Oui	Impur
Haskell	Non	Oui	Non	Oui	Non	Pur