Plan

Fonctions récursives

Une fonction récursive est une fonction

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

qui est intuitivement calculable.

Fonctions récursives

Une fonction récursive est une fonction

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

qui est intuitivement calculable.

Par exemple,

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

sont des fonctions récursives.

Fonctions récursives

Une fonction récursive est une fonction

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

qui est intuitivement calculable.

Par exemple,

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

sont des fonctions récursives.

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.

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.

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 fg;
3. soit l'**abstraction** d'une expression f, notée $\lambda x.f$ où x est une variable.

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 fg;
3. soit l'**abstraction** d'une expression f, notée $\lambda x.f$ où x est une variable.

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

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 fg;
3. soit l'**abstraction** d'une expression f, notée $\lambda x.f$ où x est une variable.

Par 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.

Plan

Plan

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.

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.

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**.

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.

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.

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.

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.

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

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

Transparence référentielle

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

Considérons le code C suivant :

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

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

Transparence référentielle

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

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)`.

Transparence référentielle

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

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.

Transparence référentielle

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

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 l'on fera

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

Ce que l'on fera

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).

Ce que l'on fera

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**.

Ce que l'on fera

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

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 ».

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 ;

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

Considérons le programme Python

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

Vérification de types dynamique

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 ;

Vérification de types dynamique

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.

Vérification de types dynamique

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.

Vérification de types dynamique

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

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;
}
```

Vérification de types statique

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 *`.

Vérification de types statique

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.

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).

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.

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).

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 utilisées dans un programme :

1. attribution explicite, où les **types** des variables sont **mentionnés** dans le programme ;

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 utilisées dans un programme :

1. attribution explicite, où les **types** des variables sont **mentionnés** dans le programme ;

2. attribution implicite, où les **types** des variables 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.

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 utilisées dans un programme :

1. attribution explicite, où les **types** des variables sont **mentionnés** dans le programme ;

2. attribution implicite, où les **types** des variables 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

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;
}
```

Attribution explicite

Considérons le programme C

```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.

Attribution explicite

Considérons le programme C

```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

Considérons le programme Caml

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

Attribution implicite

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.

Attribution implicite

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.

Attribution implicite

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.

Avantages et inconvénients

Attribution explicite.

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

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.

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.

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 ;

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

Considérons le programme Caml

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

Portée statique

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

Considérons le programme Python

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

Portée dynamique

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"` ;

Portée dynamique

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 contenant successivement les éléments `1`, `2` et `3` ;

Portée dynamique

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 contenant successivement les éléments `1`, `2` et `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*).

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	V. dyn.	V. 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

Caractéristiques des principaux langages

Langage	V. dyn.	V. 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**	**Non**	**Impur**
Haskell	Non	Oui	Non	Oui	Non	Pur

Axe 2 : concepts premiers

Programmation

Pratique

Types

Plan

La 1re chose à apprendre

En CAML, un commentaire est constitué de tout ce qui est délimité par `(*` et `*)`. Par exemple,

```
(* Ceci est un commentaire. *).
```

La 1re chose à apprendre

En CAML, un commentaire est constitué de tout ce qui est délimité par `(*` et `*)`. Par exemple,

```
(* Ceci est un commentaire. *).
```

Les symboles `(*` et `*)` **fonctionnent comme des parenthèses** : il est possible d'appareiller chaque `(*` avec un unique `*)`. Il est ainsi possible d'imbriquer les commentaires.

La 1re chose à apprendre

En CAML, un commentaire est constitué de tout ce qui est délimité par `(*` et `*)`. Par exemple,

```
(* Ceci est un commentaire. *).
```

Les symboles `(*` et `*)` **fonctionnent comme des parenthèses** : il est possible d'appareiller chaque `(*` avec un unique `*)`. Il est ainsi possible d'imbriquer les commentaires.

```
(* Ceci est un commentaire
(* imbrique. *) *)
```

En CAML, ceci fonctionne.

La 1re chose à apprendre

En CAML, un commentaire est constitué de tout ce qui est délimité par `(*` et `*)`. Par exemple,

```
(* Ceci est un commentaire. *).
```

Les symboles `(*` et `*)` **fonctionnent comme des parenthèses** : il est possible d'appareiller chaque `(*` avec un unique `*)`. Il est ainsi possible d'imbriquer les commentaires.

```
(* Ceci est un commentaire
(* imbrique. *) *)
```

En CAML, ceci fonctionne.

```
/* Ceci est un commentaire
/* imbrique. */ */
```

En C, ceci ne fonctionne pas.

Plan

Différents modes d'utilisation

Il existe trois modes principaux pour programmer en Caml :

Différents modes d'utilisation

Il existe trois modes principaux pour programmer en CAML :

1. écrire un programme dans un fichier d'extension `.ml` et le compiler avec le **compilateur bytecode** ;

Différents modes d'utilisation

Il existe trois modes principaux pour programmer en CAML :

1. écrire un programme dans un fichier d'extension `.ml` et le compiler avec le **compilateur bytecode** ;
2. écrire un programme dans un fichier d'extension `.ml` et le compiler avec le **compilateur natif** ;

Différents modes d'utilisation

Il existe trois modes principaux pour programmer en CAML :

1. écrire un programme dans un fichier d'extension `.ml` et le compiler avec le **compilateur bytecode** ;
2. écrire un programme dans un fichier d'extension `.ml` et le compiler avec le **compilateur natif** ;
3. ouvrir l'**interpréteur** CAML et écrire du code, ou bien en importer.

Différents modes d'utilisation

Il existe trois modes principaux pour programmer en CAML :

1. écrire un programme dans un fichier d'extension `.ml` et le compiler avec le **compilateur bytecode** ;
2. écrire un programme dans un fichier d'extension `.ml` et le compiler avec le **compilateur natif** ;
3. ouvrir l'**interpréteur** CAML et écrire du code, ou bien en importer.

Les deux premières méthodes fournissent un exécutable (voir plus loin).

Différents modes d'utilisation

Il existe trois modes principaux pour programmer en CAML :

1. écrire un programme dans un fichier d'extension `.ml` et le compiler avec le **compilateur bytecode** ;
2. écrire un programme dans un fichier d'extension `.ml` et le compiler avec le **compilateur natif** ;
3. ouvrir l'**interpréteur** CAML et écrire du code, ou bien en importer.

Les deux premières méthodes fournissent un exécutable (voir plus loin).

La 3e est intéressante car elle permet de faire du développement incrémental, c.-à-d. l'écriture et le test pas à pas des fonctions nécessaires à la résolution d'un problème.

Dans ce cas, le programme n'est pas exécuté mais est interprété.

Interpréteur

L'interpréteur se lance avec la commande `ocaml` ou mieux, `rlwrap ocaml` pour avoir accès à l'historique.

```
        Objective Caml version 4.09.0

#
```

Interpréteur

L'interpréteur se lance avec la commande `ocaml` ou mieux, `rlwrap ocaml` pour avoir accès à l'historique.

```
        Objective Caml version 4.09.0

#
```

Celle-ci lance une boucle d'interaction qui se comporte, de manière répétée, de la façon suivante :

Interpréteur

L'interpréteur se lance avec la commande `ocaml` ou mieux, `rlwrap ocaml` pour avoir accès à l'historique.

```
        Objective Caml version 4.09.0

#
```

Celle-ci lance une boucle d'interaction qui se comporte, de manière répétée, de la façon suivante :

1. le programmeur **écrit** une phrase ;

Lire

Interpréteur

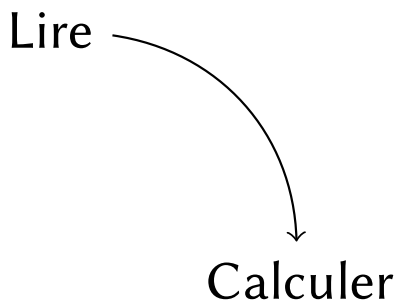
L'interpréteur se lance avec la commande `ocaml` ou mieux, `rlwrap ocaml` pour avoir accès à l'historique.

```
        Objective Caml version 4.09.0

#
```

Celle-ci lance une boucle d'interaction qui se comporte, de manière répétée, de la façon suivante :

1. le programmeur **écrit** une phrase ;
2. le système l'**interprète** ;



Interpréteur

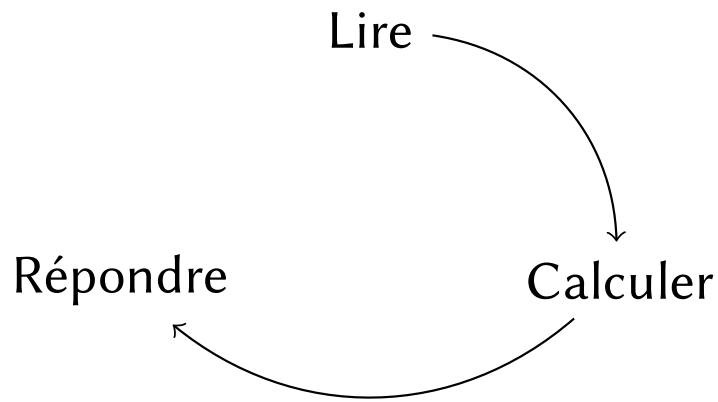
L'interpréteur se lance avec la commande `ocaml` ou mieux, `rlwrap ocaml` pour avoir accès à l'historique.

```
        Objective Caml version 4.09.0

#
```

Celle-ci lance une boucle d'interaction qui se comporte, de manière répétée, de la façon suivante :

1. le programmeur **écrit** une phrase ;
2. le système l'**interprète** ;
3. le système affiche le **résultat** de la phrase.



Interpréteur

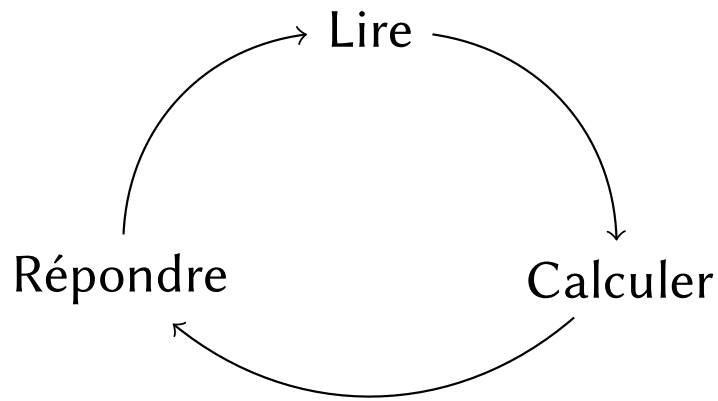
L'interpréteur se lance avec la commande `ocaml` ou mieux, `rlwrap ocaml` pour avoir accès à l'historique.

```
        Objective Caml version 4.09.0

#
```

Celle-ci lance une boucle d'interaction qui se comporte, de manière répétée, de la façon suivante :

1. le programmeur **écrit** une phrase ;
2. le système l'**interprète** ;
3. le système affiche le **résultat** de la phrase.



Et ainsi de suite.

Phrases et réponses

Une phrase est une **expression** terminée par `;;` (marqueur de fin de phrase).

```
# 1 + 1;;
```

Phrases et réponses

Une phrase est une **expression** terminée par `;;` (marqueur de fin de phrase).

```
# 1 + 1;;
```

Elle peut tenir sur plusieurs lignes

```
# 1
  + 1;;
```

Phrases et réponses

Une phrase est une **expression** terminée par `;;` (marqueur de fin de phrase).

```
# 1 + 1;;
```

Elle peut tenir sur plusieurs lignes

```
# 1
  + 1;;
```

L'utilisateur demande l'**évaluation** d'une phrase en appuyant sur **entrée** et le système fournit ensuite sa réponse.

```
# 1 + 1;;
- : int = 2
```

Phrases et réponses

Une phrase est une **expression** terminée par `;;` (marqueur de fin de phrase).

```
# 1 + 1;;
```

Elle peut tenir sur plusieurs lignes

```
# 1
  + 1;;
```

L'utilisateur demande l'**évaluation** d'une phrase en appuyant sur **entrée** et le système fournit ensuite sa réponse.

```
# 1 + 1;;
- : int = 2
```

Explications :

- le signe `-` signifie qu'une **valeur** a été calculée ;

Phrases et réponses

Une phrase est une **expression** terminée par `;;` (marqueur de fin de phrase).

```
# 1 + 1;;
```

Elle peut tenir sur plusieurs lignes

```
# 1
  + 1;;
```

L'utilisateur demande l'**évaluation** d'une phrase en appuyant sur **entrée** et le système fournit ensuite sa réponse.

```
# 1 + 1;;
- : int = 2
```

Explications :

- le signe `-` signifie qu'une **valeur** a été calculée ;
- `: int` signifie que cette valeur est de **type** `int` ;

Phrases et réponses

Une phrase est une **expression** terminée par `;;` (marqueur de fin de phrase).

```
# 1 + 1;;
```

Elle peut tenir sur plusieurs lignes

```
# 1
  + 1;;
```

L'utilisateur demande l'**évaluation** d'une phrase en appuyant sur **entrée** et le système fournit ensuite sa réponse.

```
# 1 + 1;;
- : int = 2
```

Explications :

- le signe `-` signifie qu'une **valeur** a été calculée ;
- `: int` signifie que cette valeur est de **type** `int` ;
- `= 2` signifie que cette valeur **est** `2`.

Plan

Nommer une valeur

De la même manière que l'on peut écrire en mathématiques

« *Soit* n *l'entier* 5. »,

pour définir ce que représente le symbole « n », il est possible en programmation CAML de donner un nom à une valeur.

Nommer une valeur

De la même manière que l'on peut écrire en mathématiques

$$\text{« } \textit{Soit } n \textit{ l'entier } 5. \text{ »,}$$

pour définir ce que représente le symbole « n », il est possible en programmation CAML de donner un nom à une valeur.

Ceci s'appelle une définition, ou encore la liaison d'un nom à une valeur.

Nommer une valeur

De la même manière que l'on peut écrire en mathématiques

$$\text{« } \textit{Soit } n \textit{ l'entier } 5. \text{ »,}$$

pour définir ce que représente le symbole « n », il est possible en programmation CAML de donner un nom à une valeur.

Ceci s'appelle une définition, ou encore la liaison d'un nom à une valeur.

On utilise pour cela la construction syntaxique

```
let ID = VAL
```

où `ID` est un nom (identificateur) et `VAL` est une expression.

Nommer une valeur

De la même manière que l'on peut écrire en mathématiques

$$\text{« } \textit{Soit } n \textit{ l'entier } 5. \text{ »,}$$

pour définir ce que représente le symbole « n », il est possible en programmation CAML de donner un nom à une valeur.

Ceci s'appelle une définition, ou encore la liaison d'un nom à une valeur.

On utilise pour cela la construction syntaxique

```
let ID = VAL
```

où `ID` est un nom (identificateur) et `VAL` est une expression.

```
# let n = 5;;
val n : int = 5
```

La 1re phrase lie au nom `n` la valeur `5`. L'interpréteur **le signale** en commençant sa réponse par `val n`.

Nommer une valeur

De la même manière que l'on peut écrire en mathématiques

$$\text{« } \textit{Soit } n \textit{ l'entier } 5. \text{ »,}$$

pour définir ce que représente le symbole « n », il est possible en programmation CAML de donner un nom à une valeur.

Ceci s'appelle une définition, ou encore la liaison d'un nom à une valeur.

On utilise pour cela la construction syntaxique

```
let ID = VAL
```

où `ID` est un nom (identificateur) et `VAL` est une expression.

```
# let n = 5;;
val n : int = 5
```

La 1re phrase lie au nom `n` la valeur `5`. L'interpréteur **le signale** en commençant sa réponse par `val n`.

```
# n;;
- : int = 5
```

La 2e phrase donne la valeur à laquelle `n` est liée.

Liaisons locales

Il est possible de définir des noms localement à une expression.

Liaisons locales

Il est possible de définir des noms localement à une expression.

La **portée lexicale** d'un nom défini localement est étendue à l'expression où figure sa définition.

Liaisons locales

Il est possible de définir des noms localement à une expression.

La **portée lexicale** d'un nom défini localement est étendue à l'expression où figure sa définition.

On utilise pour cela la construction syntaxique

```
let ID = VAL in EXP
```

où `ID` est un nom, `VAL` est une expression et `EXP` est une expression.

Liaisons locales

Il est possible de définir des noms localement à une expression.

La **portée lexicale** d'un nom défini localement est étendue à l'expression où figure sa définition.

On utilise pour cela la construction syntaxique

```
let ID = VAL in EXP
```

où `ID` est un nom, `VAL` est une expression et `EXP` est une expression.

Très important : cette expression possède comme valeur celle de `EXP` dans laquelle les occurrences libres de `ID` sont remplacées par `VAL`.

Liaisons locales

Il est possible de définir des noms localement à une expression.

La **portée lexicale** d'un nom défini localement est étendue à l'expression où figure sa définition.

On utilise pour cela la construction syntaxique

```
let ID = VAL in EXP
```

où `ID` est un nom, `VAL` est une expression et `EXP` est une expression.

Très important : cette expression possède comme valeur celle de `EXP` dans laquelle les occurrences libres de `ID` sont remplacées par `VAL`.

```
# let n = 5 in n + 1;;
- : int = 6
```

La 2e occurrence de `n` a pour valeur `5` à cause de la liaison locale précédente. Ainsi, `n + 1` a pour valeur `6`.

Liaisons locales

Il est possible de définir des noms localement à une expression.

La **portée lexicale** d'un nom défini localement est étendue à l'expression où figure sa définition.

On utilise pour cela la construction syntaxique

```
let ID = VAL in EXP
```

où `ID` est un nom, `VAL` est une expression et `EXP` est une expression.

Très important : cette expression possède comme valeur celle de `EXP` dans laquelle les occurrences libres de `ID` sont remplacées par `VAL`.

```
# let n = 5 in n + 1;;
- : int = 6
```

La 2e occurrence de `n` a pour valeur `5` à cause de la liaison locale précédente. Ainsi, `n + 1` a pour valeur `6`.

```
# let n = 3;;
val n : int = 3
# let n = 4 in 2 * n;;
- : int = 8
# n;;
- : int = 3
```

La liaison de `n` à `4` dans la 2e phrase est locale : le nom global `n` défini en 1re phrase reste inchangé.

Liaisons locales

Il est possible de définir des noms localement à une expression.

La **portée lexicale** d'un nom défini localement est étendue à l'expression où figure sa définition.

On utilise pour cela la construction syntaxique

```
let ID = VAL in EXP
```

où `ID` est un nom, `VAL` est une expression et `EXP` est une expression.

Très important : cette expression possède comme valeur celle de `EXP` dans laquelle les occurrences libres de `ID` sont remplacées par `VAL`.

```
# let n = 5 in n + 1;;
- : int = 6
```

La 2e occurrence de `n` a pour valeur `5` à cause de la liaison locale précédente. Ainsi, `n + 1` a pour valeur `6`.

```
# let n = 3;;
val n : int = 3
# let n = 4 in 2 * n;;
- : int = 8
# n;;
- : int = 3
```

La liaison de `n` à `4` dans la 2e phrase est locale : le nom global `n` défini en 1re phrase reste inchangé.

Ceci renseigne sur la valeur du nom `n` de la 3e phrase.

Liaisons locales — exemples

La règle pour comprendre une phrase définissant un nom est de **relier** chaque occurrence d'un nom `x` à sa définition.

Liaisons locales — exemples

La règle pour comprendre une phrase définissant un nom est de **relier** chaque occurrence d'un nom `x` à sa définition.

```
# let s = 2 in
      let s = s * s in
          let s = s * s in
              s + 4;;
```

Liaisons locales — exemples

La règle pour comprendre une phrase définissant un nom est de **relier** chaque occurrence d'un nom `x` à sa définition.

```
# let s = 2 in
      let s = s * s in
          let s = s * s in
              s + 4;;
- : int = 20
```

Chaque occurrence du nom `s` fait référence à la valeur du nom `s` de la liaison précédente. On retrouve ainsi le résultat affiché.

Liaisons locales — exemples

La règle pour comprendre une phrase définissant un nom est de **relier** chaque occurrence d'un nom `x` à sa définition.

```
# let s = 2 in
      let s = s * s in
          let s = s * s in
              s + 4;;
- : int = 20

# let s = let x = 3 in
      x * x;;
```

Chaque occurrence du nom `s` fait référence à la valeur du nom `s` de la liaison précédente. On retrouve ainsi le résultat affiché.

Liaisons locales — exemples

La règle pour comprendre une phrase définissant un nom est de **relier** chaque occurrence d'un nom `x` à sa définition.

```
# let s = 2 in
      let s = s * s in
          let s = s * s in
              s + 4;;
- : int = 20
```

Chaque occurrence du nom `s` fait référence à la valeur du nom `s` de la liaison précédente. On retrouve ainsi le résultat affiché.

```
# let s = let x = 3 in
      x * x;;
val s : int = 9
```

Le nom `s` est lié à la valeur `9`.

En effet, la sous-expression `let x = 3 in x * x` a pour valeur `9`.

Liaisons locales — exemples

```
# let x =
      let y = 2 in
          let z = 3 in
              y + let z = 8 in
                  z * z;;
```

Liaisons locales — exemples

```
# let x =
      let y = 2 in
          let z = 3 in
              y + let z = 8 in
                  z * z;;
Warning 26: unused variable z.
val x : int = 66
```

La nom `z` défini en l. 3 n'est pas utilisé pour attribuer une valeur au nom `x`.

L'interpréteur le signale par un message d'avertissement.

Liaisons locales — exemples

```
# let x =
      let y = 2 in
          let z = 3 in
              y + let z = 8 in
                  z * z;;
Warning 26: unused variable z.
val x : int = 66

# let x = let y = 2 in
      x + y;;
```

La nom `z` défini en l. 3 n'est pas utilisé pour attribuer une valeur au nom `x`.

L'interpréteur le signale par un message d'avertissement.

Liaisons locales — exemples

```
# let x =
      let y = 2 in
          let z = 3 in
              y + let z = 8 in
                  z * z;;
Warning 26: unused variable z.
val x : int = 66
```

La nom `z` défini en l. 3 n'est pas utilisé pour attribuer une valeur au nom `x`.

L'interpréteur le signale par un message d'avertissement.

```
# let x = let y = 2 in
      x + y;;
Error: Unbound value x
```

L'évaluation de cette phrase produit une erreur lors de son interprétation. En effet, le nom `x` de la l. 2 n'est pas défini.

Liaisons (locales) simultanées

Il est possible de définir des noms ou de réaliser des liaisons locales simultanément dans une même phrase.

Liaisons (locales) simultanées

Il est possible de définir des noms ou de réaliser des liaisons locales simultanément dans une même phrase.

On utilise pour cela les constructions syntaxiques

```
let ID1 = VAL1 and ID2 = VAL2
```

ou

```
let ID1 = VAL1 and ID2 = VAL2 in EXP
```

où `ID1` et `ID2` sont des identificateurs, et `VAL1`, `VAL2` et `EXP` sont des expressions.

Liaisons (locales) simultanées

Il est possible de définir des noms ou de réaliser des liaisons locales simultanément dans une même phrase.

On utilise pour cela les constructions syntaxiques

```
let ID1 = VAL1 and ID2 = VAL2
```

ou

```
let ID1 = VAL1 and ID2 = VAL2 in EXP
```

où `ID1` et `ID2` sont des identificateurs, et `VAL1`, `VAL2` et `EXP` sont des expressions.

```
# let x = 1 and y = 2;;
val x : int = 1
val y : int = 2
```

Cette phrase lie simultanément au nom `x` la valeur `1` et au nom `y` la valeur `2`.

Liaisons (locales) simultanées

Il est possible de définir des noms ou de réaliser des liaisons locales simultanément dans une même phrase.

On utilise pour cela les constructions syntaxiques

```
let ID1 = VAL1 and ID2 = VAL2
```

ou

```
let ID1 = VAL1 and ID2 = VAL2 in EXP
```

où `ID1` et `ID2` sont des identificateurs, et `VAL1`, `VAL2` et `EXP` sont des expressions.

```
# let x = 1 and y = 2;;
val x : int = 1
val y : int = 2
```

Cette phrase lie simultanément au nom `x` la valeur `1` et au nom `y` la valeur `2`.

```
# let x = 1 in
      let y = 3
      and z = 4 in
          x + y + z;;
- : int = 8
```

Il est possible d'imbriquer les définitions locales et simultanées.

On notera l'indentation différente impliquée par les `in` et les `and`.

Liaisons simultanées — exemples

```
# let x = 1 in
      let x = 2
      and y = x + 3 in
          x + y;;
```

Liaisons simultanées — exemples

```
# let x = 1 in
      let x = 2
      and y = x + 3 in
          x + y;;
- : int = 6
```

Dans la définition du nom `y`, l'occurrence de `x` qui y apparaît est celle définie en l. 1.

Liaisons simultanées — exemples

```
# let x = 1 in
      let x = 2
      and y = x + 3 in
          x + y;;
- : int = 6

# let x = 1 in
      let x = 2 in
          let y = x + 3 in
              x + y;;
```

Dans la définition du nom `y`, l'occurrence de `x` qui y apparaît est celle définie en l. 1.

Liaisons simultanées — exemples

```
# let x = 1 in
      let x = 2
      and y = x + 3 in
          x + y;;
- : int = 6
```

Dans la définition du nom `y`, l'occurrence de `x` qui y apparaît est celle définie en l. 1.

```
# let x = 1 in
      let x = 2 in
          let y = x + 3 in
              x + y;;
Warning 26: unused variable x.
- : int = 7
```

Cette phrase est obtenue en remplaçant le `and` par un `in let`.

Liaisons simultanées — exemples

```
# let x = 1 in
      let x = 2
      and y = x + 3 in
          x + y;;
- : int = 6
```

Dans la définition du nom `y`, l'occurrence de `x` qui y apparaît est celle définie en l. 1.

```
# let x = 1 in
      let x = 2 in
          let y = x + 3 in
              x + y;;
Warning 26: unused variable x.
- : int = 7
```

Cette phrase est obtenue en remplaçant le `and` par un `in let`.

Le résultat est différent du précédent : dans la définition du nom `y`, l'occurrence de `x` qui y apparaît est celle définie en l. 2.

Liaisons simultanées — exemples

```
# let x = 1
  and y = 2
  and z = let x = 16 in
          x * x * x;;
```

Liaisons simultanées — exemples

```
# let x = 1
  and y = 2
  and z = let x = 16 in
      x * x * x;;
val x : int = 1
val y : int = 2
val z : int = 4096
```

Il y a trois liaisons simultanées.

Liaisons simultanées — exemples

```
# let x = 1
  and y = 2
  and z = let x = 16 in
          x * x * x;;
val x : int = 1
val y : int = 2
val z : int = 4096
```

Il y a trois liaisons simultanées.

L'occurrence du nom `x` en l. 4 est celle définie en l. 3. Cette définition n'influe pas sur la définition de `x` en l. 1.

Liaisons simultanées — exemples

```
# let x = 1
  and y = 2
  and z = let x = 16 in
          x * x * x;;
val x : int = 1
val y : int = 2
val z : int = 4096

# let x = 1
  and y = x + 1;;
```

Il y a trois liaisons simultanées.

L'occurrence du nom `x` en l. 4 est celle définie en l. 3. Cette définition n'influe pas sur la définition de `x` en l. 1.

Liaisons simultanées — exemples

```
# let x = 1
  and y = 2
  and z = let x = 16 in
          x * x * x;;
val x : int = 1
val y : int = 2
val z : int = 4096
```

Il y a trois liaisons simultanées.

L'occurrence du nom `x` en l. 4 est celle définie en l. 3. Cette définition n'influe pas sur la définition de `x` en l. 1.

```
# let x = 1
  and y = x + 1;;
Error: Unbound value x
```

L'évaluation de cette phrase produit une erreur lors de son interprétation. En effet, le nom `x` de la l. 2 n'est pas défini.

Différence entre liaisons et liaisons locales

Les liaisons `let ID = VAL` et les liaisons locales `let ID = VAL in EXP` sont des entités totalement différentes.

Différence entre liaisons et liaisons locales

Les liaisons `let ID = VAL` et les liaisons locales `let ID = VAL in EXP` sont des entités totalement différentes.

En effet,

- une liaison `let ID = VAL` permet de former une phrase qui n'a pas de valeur et dont le but est de créer un alias `ID` pour la valeur de `VAL` dans la suite du programme.

Différence entre liaisons et liaisons locales

Les liaisons `let ID = VAL` et les liaisons locales `let ID = VAL in EXP` sont des entités totalement différentes.

En effet,

- une liaison `let ID = VAL` permet de former une phrase qui n'a pas de valeur et dont le but est de créer un alias `ID` pour la valeur de `VAL` dans la suite du programme. P.ex.,

```
# let x = 3;;
val x : int = 3
...
# x + 1;;
- : int = 4
```

Différence entre liaisons et liaisons locales

Les liaisons `let ID = VAL` et les liaisons locales `let ID = VAL in EXP` sont des entités totalement différentes.

En effet,

- une liaison `let ID = VAL` permet de former une phrase qui n'a pas de valeur et dont le but est de créer un alias `ID` pour la valeur de `VAL` dans la suite du programme. P.ex.,

```
# let x = 3;;
val x : int = 3
...
# x + 1;;
- : int = 4
```

- Une liaison locale `let ID = VAL in EXP` permet de former une phrase qui a une valeur et qui est celle de `EXP` dans laquelle les occurrences de `ID` sont remplacées par la valeur de `VAL`.

Différence entre liaisons et liaisons locales

Les liaisons `let ID = VAL` et les liaisons locales `let ID = VAL in EXP` sont des entités totalement différentes.

En effet,

- une liaison `let ID = VAL` permet de former une phrase qui n'a pas de valeur et dont le but est de créer un alias `ID` pour la valeur de `VAL` dans la suite du programme. P.ex.,

```
# let x = 3;;
val x : int = 3
...
# x + 1;;
- : int = 4
```

- Une liaison locale `let ID = VAL in EXP` permet de former une phrase qui a une valeur et qui est celle de `EXP` dans laquelle les occurrences de `ID` sont remplacées par la valeur de `VAL`. P.ex.,

```
# let x = 3 in x + 1;;
- : int = 4
```