
Programmation fonctionnelle et logique

INF6120

Samuele Giraudo

UQÀM

Automne 2026

1. Introduction	3
2. Notions générales	57
3. Programmation fonctionnelle	117
4. Programmation logique	438

1. Introduction

1.1. Informations générales 5

1.2. Introduction 17

1.3. Initiation rapide à l'OCaml 35

1.1. Informations générales

1.1.1. Informations pratiques 7

1.1.2. Aperçu 11

1.1.3. Références 14

1.1.1. Informations pratiques

- Titre du cours : *Programmation fonctionnelle et logique*
- Sigle : INF6120
- Département : Informatique
- Coordinateurs : Samuele Giraudo
- Enseignant : Samuele Giraudo
- Courriel : giraudo.samuele@uqam.ca
- Bureau : PK-4430
- Page personnelle : <https://www.giraudo.uqam.ca>
- Site du cours : <https://www.giraudo.uqam.ca/Teaching/INF6120/2026-09/INF6120.html>
- Plan de cours : https://info.uqam.ca/plan_cours/Automne%202026/INF6120.html

Prérequis :

- validation d'INF3105, *Structures de données et algorithmes* ;

ainsi que toutes ses dépendances :

- validation d'INF1120, *Programmation I* ;
- validation d'INF1132, *Mathématiques pour l'informatique* ou de MAT1060, *Mathématiques algorithmiques* ;
- validation d'INF2120, *Programmation II*.

Calendrier des évaluations avec leurs pondérations :

1. séance 5 : examen 1, 33 % ;
2. séance 10 : examen 2, 33 % ;
3. séance 15 : examen 3, 34 %.

Contenu général :

- l'examen 1 porte sur ce qui a été vu dans les 4 premières séances de cours (programmation fonctionnelle) ;
- l'examen 2 porte sur ce qui a été vu dans les 8 premières séances de cours (programmation fonctionnelle) ;
- l'examen 3 porte sur ce qui a été vu dans toutes les 12 premières séances de cours (programmation fonctionnelle et programmation logique).

Pour les examens, aucun document n'est autorisé. Tout appareil électronique est interdit.

Pour valider le cours, il faut avoir une moyenne supérieure à 50 %.

1.1.2. Aperçu

Objectifs principaux :

- connaître les principes généraux du **paradigme de programmation fonctionnel** ;
- connaître les principes généraux du **paradigme de programmation logique** ;
- savoir **développer des applications en OCaml** ;
- savoir **développer des applications en Prolog**.

Ce n'est pas un cours d'apprentissage de langages, c'est un cours de **découverte de paradigmes**.

Le désavantage à cela est que la connaissance spécifique de ces deux langages sera relativement modérée.

L'avantage est que cela permet très facilement d'apprendre d'autres langages s'inscrivant dans des paradigmes similaires.

Le cours est organisé en deux parties de volumes respectifs environ 75 % / 25 %.

1. Programmation fonctionnelle.

- Bases théoriques.
- Machines de Turing et λ -calcul.
- Caractéristiques principales des langages de programmation.
- Programmation en OCaml.
- Principes de programmation fonctionnelle.
- Démonstration de justesse de programmes.
- Stratégies d'évaluation.

2. Programmation logique.

- Bases théoriques.
- Unification et résolution.
- Manipulation des listes.

1.1.3. Références

Les deux **références principales** suivantes couvrent une grande partie de ce cours.

1. M. R. Clarkson, OCaml Programming: Correct + Efficient + Beautiful, Fall 2026 Edition, 2026, <https://cs3110.github.io/textbook/cover.html>.
2. P. Flach, Simply Logical - Intelligent Reasoning by Example (Fully Interactive Online Edition), 2022, <https://book.simply-logical.space/src/simply-logical.html>.

Elles sont consultables librement aux liens indiqués.

Quelques **références secondaires** utiles et très intéressantes :

1. Y. Minsky, A. Madhavapeddy, Real World OCaml, 2nd Edition, 2022,
<https://dev.realworldocaml.org/>.
2. X. Leroy *et al.*, The OCaml system release 5.5, 2026,
<https://ocaml.org/manual/5.5/index.html>.

Quelques **autres références**, pour approfondir, mais non obligatoires :

1. G. Dowek, J.-J. Lévy, Introduction à la théorie des langages de programmation, École Polytechnique, 2006.
2. C. Okasaki, Purely Functional Data Structures, Cambridge University Press, 1999.
3. S. L. Peyton Jones, D. Lester, Implementing Functional Languages, Prentice Hall, 1992.
4. P. Hudak, D. Quick, The Haskell School of Music: From Signals to Symphonies, Cambridge University Press, 2018.

1.2. Introduction

- 1.2.1. Arbres binaires de recherche en programmation fonctionnelle 19
- 1.2.2. Instructions vs expressions 24
- 1.2.3. Principes généraux de la programmation fonctionnelle 27
- 1.2.4. Fonctions vs prédicats 30
- 1.2.5. Principes généraux de la programmation logique 33

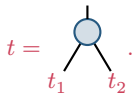
1.2.1. Arbres binaires de recherche en programmation fonctionnelle

Un **arbre binaire** t est

1. soit une feuille :

$$t = \text{!}$$

2. soit un nœud attaché à deux arbres binaires t_1 et t_2 :



C'est une définition **récursive** car la définition de la notion fait référence à la notion qui est en cours de définition.

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

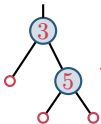
En OCaml, ceci peut se faire via la **définition du type**

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

L'expression

```
Node (Leaf, 3, Node (Leaf, 5, Leaf))
```

est de type `binary_trees` et désigne l'arbre binaire



Les entités `Leaf` et `Node` permettent de construire des données du type `binary_trees`. Ce sont des **constructeurs** de ce type.

L'algorithme d'insertion dans un arbre binaire de recherche admet la description suivante.

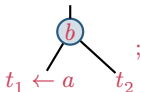
Soit t un arbre binaire de recherche et a une valeur.

L'arbre $t \leftarrow a$, obtenu en insérant a dans t , est défini par :

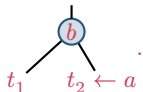
1. si t est une feuille, alors $t \leftarrow a :=$  ;

2. sinon, t est de la forme $t =$  où b est une valeur. Dans ce cas :

• si $a \leq b$, alors

$t \leftarrow a :=$  ;

• sinon, $a > b$ et

$t \leftarrow a :=$  .

En OCaml, l'algorithme précédent peut s'implanter via la **définition de fonction**

```
let rec insert t a =  
  match t with  
  | Leaf -> Node (Leaf, a, Leaf)  
  | Node (t1, b, t2) when a <= b -> Node (insert t1 a, b, t2)  
  | Node (t1, b, t2) -> Node (t1, b, insert t2 a)
```

La **construction** d'un arbre binaire de recherche peut se faire par insertions successives :

```
let t0 = insert Leaf 3 in  
let t1 = insert t0 4 in  
let t2 = insert t1 2 in  
let t3 = insert t2 3 in  
let t4 = insert t3 4 in  
let t5 = insert t4 1 in  
t5
```

Toute cette expression s'**évalue** en une **valeur** de **type** `binary_trees`. Il s'agit d'un arbre binaire de recherche ayant 5 nœuds internes.

Nous verrons plus tard qu'il existe des manières beaucoup plus concises pour écrire ces sortes de définitions.

1.2.2. Instructions vs expressions

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**, ceci peut se faire en Python via la fonction

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

Le calcul de `one_of_two([0, 1, 2, 3, 4])` utilise de la **mémoire** annexe

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

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

Sous le paradigme fonctionnel, dans un OCaml de débutant, nous pouvons considérer la fonction

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

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

`one_of_two [0; 1; 2; 3; 4] → 0 :: one_of_two [2; 3; 4] → 0 :: 2 :: one_of_two [4] → 0 :: 2 :: [4] ⇒ [0; 2; 4]`.

Dans ce cas présent, 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. Les définitions de **fonctions** permettent d'expliquer comment réaliser un calcul.

L'expression est en mémoire, mais il n'y a pas besoin d'enregistrer des données annexes.

1.2.3. Principes généraux de la programmation fonctionnelle

Quelques caractéristiques de la programmation fonctionnelle (en OCaml spécifiquement) :

- Allocation de mémoire *implicite*.

Le programmeur se concentre sur des aspects moins techniques et plus importants.

- Programmer = *penser*.

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.

En pratique :

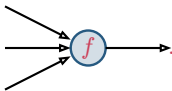
« Le compilateur OCaml n'accepte
presque jamais mon code mais quand il
l'accepte, ça marche *presque toujours*
comme je le veux. »

vs

« Le compilateur X accepte *presque*
toujours mon code mais ça ne marche
presque jamais comme je le veux. »

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

Une fonction f est une entité qui accepte des valeurs en entrée et qui produit une valeur en sortie :



On **évite les effets secondaires** (et donc, en particulier, on ne fait pas d'affectation). Les fonctions ne font rien d'autre que de renvoyer des valeurs.

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

On utilise en revanche (beaucoup) la **définition de fonctions**, l'**application de fonctions** et la **récurtivité**.

1.2.4. Fonctions vs prédicats

Considérons le problème qui, étant donné une liste `lst` et une valeur `x` en entrées, consiste à connaître la position d'une occurrence de `x` dans `lst`.

Sous le **paradigme impératif**, ceci peut se faire en Python via la fonction

```
def first(lst, x):  
    i = 0  
    for i in range(len(lst)):  
        if lst[i] == x:  
            return i  
    return -1
```

Sous le **paradigme fonctionnel**, ceci peut se faire en OCaml via la fonction

```
let first lst x =  
    let rec aux lst i =  
        match lst with  
        | [] -> -1  
        | x' :: lst' when x = x' -> i  
        | _ :: lst' -> aux lst' (i + 1)  
    in  
    aux lst 0
```

Ces deux fonctions renvoient plus précisément la **première occurrence** de `x` dans `lst`.

La valeur de retour `-1` est utilisée pour spécifier l'absence de `x` dans `lst`.

Sous le **paradigme logique**, en Prolog, nous pouvons considérer le prédicat

```
first([X | _], X, 0).  
  
first([_ | Tail], X, Pos) :-  
    first(Tail, X, Pos_1),  
    Pos is Pos_1 + 1.
```

Ce prédicat `first(Lst, X, I)` **teste** si la valeur en l'indice `I` de la liste `Lst` est `X`.

La **résolution** de la **requête** `first([2, 10, 3, 10, 1, 1, 4, 10], 10, 1)` donne `true.`.

Nous pouvons observer que la résolution de `first([2, 10, 3, 10, 1, 1, 4, 10], 10, 3)` donne aussi `true.`

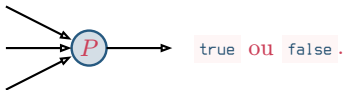
Dans ce paradigme, il est possible de laisser le troisième argument en tant que **variable**. Dans ce cas, lors de la résolution, **toutes les solutions** vont être engendrées.

Ainsi, la requête `first([2, 10, 3, 10, 1, 1, 4, 10], 10, I)` donne `I = 1 ; I = 3 ; I = 7 ; false.`

Ceci constitue un avantage énorme de ce paradigme.

1.2.5. Principes généraux de la programmation logique

La **programmation logique** est un paradigme de programmation dans lequel les objets de base sont les **clauses** et les **prédicats**. Un prédicat P est une entité qui accepte des valeurs en entrée et qui répond par **true** ou **false** :



Les principales **caractéristiques du paradigme fonctionnel** (absence d'effet secondaire, absence de boucle, utilisation de la récursivité) s'appliquent aussi au paradigme logique.

Il n'y a plus de notion de **sortie** aussi claire que dans les autres paradigmes : chaque entrée d'un prédicat peut être vue comme une sortie et réciproquement.

1.3. Initiation rapide à l'OCaml

1.3.1. Programmes 37

1.3.2. Principes de base 40

1.3.3. Structures principales 43

1.3.4. Types de base 49

1.3.5. Manipulation de listes 51

1.3.1. Programmes

Un **programme OCaml** est un fichier texte d'extension `.ml`.

Il est structuré de la manière suivante :

1. suite de définitions (de types et de fonctions) ;
2. fonction principale.

La **fonction principale** est le **point d'entrée** du programme : c'est par elle que commence son exécution.

Celle-ci se définit via la syntaxe

```
let () =  
  ...
```

Les `...` dénotent une expression.

Si `Program.ml` est un programme OCaml structuré selon la description précédente, il s'exécute via la commande

```
ocaml Program.ml
```

Il est également possible d'obtenir un exécutable via la commande

```
ocamlc Program.ml
```

Par défaut, l'exécutable construit se nomme `a.out`.

D'autres façons (plus évoluées) de programmer seront vues plus tard, incluant

- l'utilisation de l'interpréteur OCaml (potentiellement avec `utop`) ;
- l'utilisation du compilateur natif (`ocamlopt`) ;
- la programmation sur plusieurs fichiers (potentiellement avec `dune`).

1.3.2. Principes de base

Nous garderons en tête, dans notre initiation à l'OCaml, les principes suivants.

1. Un **programme** est une **expression**.
2. **Exécuter** un programme consiste à l'**évaluer**.
3. Le **résultat** de l'exécution d'un programme est sa **valeur**. Celle-ci est l'expression finale obtenue à l'issue du processus d'évaluation du programme.
4. Chaque **expression** possède un **unique type**.
5. Une expression doit respecter à la fois des **contraintes syntaxiques** et des **contraintes de type**.
6. Il n'y a **pas d'effet secondaire** en programmation fonctionnelle. Seule la valeur compte.
7. Cependant, nous allons assouplir la règle précédente pour nous autoriser les effets secondaires visant à réaliser des **entrées** et des **sorties**.

Exemples

```
let () =  
    11 * 10 + 3
```

Ce programme a `113` comme valeur.

```
let () =  
    let x = 11 * 10 + 3 in  
    print_int x;  
    print_newline ()
```

Ce programme a `()` comme valeur (c'est la valeur renvoyée par `print_newline ()`).

Il a comme effet secondaire d'afficher la chaîne de caractères `113` suivie d'un retour à la ligne.

```
let () =  
    let x = read_int () in  
    let y = 2 * x in  
    print_string "double : ";  
    print_int y;  
    print_newline ()
```

Ce programme a `()` comme valeur.

Il a comme effet secondaire de lire un entier sur l'entrée standard et d'afficher la chaîne de caractères dénotant le double cet entier, suivie d'un retour à la ligne.

1.3.3. Structures principales

- **Liaison globale** d'un nom à une valeur :

```
let NAME = EXP
```

Ceci fait en sorte que dans toute la suite du programme, `NAME` soit un **alias** de l'expression `EXP`.

- **Liaison locale** d'un nom à une valeur :

```
let NAME = EXP_1 in EXP_2
```

Ceci fait en sorte que dans l'expression `EXP_2`, `NAME` soit un **alias** de l'expression `EXP_1`.

Exemples

```
let size = 32

let () =
  print_int size
```

Dans ce programme, une **liaison globale** de l'alias `size` à la valeur `32` est effectuée.

```
let () =
  let x = 14 + 2 in
  let y = x * 2 in
  print_int y
```

Dans ce programme, une **liaison locale** de l'alias `x` à la valeur `16` est effectuée dans une expression dans laquelle une **liaison locale** de l'alias `y` à la valeur `x * 2` est effectuée.

- Définition d'une fonction à plusieurs paramètres :

```
let NAME X_1 ... X_N = EXP
```

Ceci fait en sorte que dans toute la suite du programme, `NAME` soit une **fonction** qui peut être appelée avec `N` arguments (nous verrons plus loin qu'il sera dans certains cas possible et même très utile d'appeler cette fonction avec un nombre différent d'arguments).

- Définition récursive d'une fonction à plusieurs paramètres :

```
let rec NAME X_1 ... X_N = EXP
```

Ceci fait en sorte que dans toute la suite du programme, `NAME` soit une **fonction** qui peut être appelée avec `N` arguments. Il est de plus possible d'appeler `NAME` dans `EXP`. Ceci peut donc potentiellement la rendre **récursive**.

Une fonction `f` s'**applique** à des arguments `a_1`, ..., `a_n` en les y **juxtaposant** par `f a_1 ... a_n`.

Exemples

```
let sum_squares x y =  
  x * x + y * y  
  
let () =  
  let x = sum_squares 8 (sum_squares 2 4) in  
  print_int x
```

```
let rec triangle n =  
  if n = 0 then  
    0  
  else  
    n + triangle (n - 1)  
  
let () =  
  let x = triangle 10 in  
  print_int x
```

Dans ce programme une **définition** de fonction nommée `sum_squares` est effectuée.

Elle est ensuite utilisée deux fois dans la fonction principale.

Dans ce programme, une **définition de fonction récursive** nommée `triangle` est effectuée.

Elle est ensuite utilisée une fois dans la fonction principale.

- Conditionnelle :

```
if EXP_1 then EXP_2 else EXP_3
```

Cette expression s'évalue en `EXP_2` si la condition `EXP_1` est vraie et en `EXP_3` sinon.

- Filtrage de motif :

```
match EXP with  
  | PAT_1 -> EXP_1  
  ...  
  | PAT_N -> EXP_N
```

Cette expression s'évalue en `EXP_K` si `K` est le plus petit indice tel que l'expression `EXP` est filtrée par le motif `PAT_K`.

Exemples

```
if 244 mod 3 = 1 then 21 else 31
```

Cette expression s'évalue en 21 car la condition `244 mod 3 = 1` est vraie.

```
let one_or_two x =  
  match x with  
    | 1 -> 1  
    | 2 -> 2  
    | 3 -> 2  
    | 4 -> 2
```

Cette fonction implante l'application $f: \{1,2,3,4\} \rightarrow \{1,2\}$ telle que $f(1) = 1$, $f(2) = 2$, $f(3) = 2$ et $f(4) = 2$.

Par exemple, `one_or_two 3` est une expression qui s'évalue en 2.

1.3.4. Types de base

Les types de base sont

- le type `int` des **entiers**, exprimés en décimal et munis des opérateurs classiques ;
- le type `string` des **chaînes de caractères**, exprimées entre `"` ;
- le type `bool` des **booléens**, `true` et `false` ;
- le type `'a * 'b` des **couples**, exprimés entre parenthèses et séparés par une virgule.

Exemples

Quelques expressions

- de type `int` : `0`, `0 + 0`, `if 22 mod 2 = 0 then 8 + 1 else 9 * 2` ;
- de type `string` : `""`, `"10AB"`, `"ab" ^ "bba"` ;
- de type `bool` : `true`, `let b = false in b && true` ;
- de type `'a * 'b` : `(1, 2 + 2)`, `("ab", 11)`, `(1, ("ca", "ba"))`.

1.3.5. Manipulation de listes

Une **liste** est une suite finie d'éléments, tous du même type.

Le **type** d'une liste dont les éléments sont de type **T** est **T list**.

Une liste est spécifiée par la suite de ses éléments, séparés par des **;** et entre **[** et **]**.

Exemples

- `[1; 2; 4; 7]` est une liste d'entiers. Son type est `int list`.
Cette liste contient, du premier au dernier, les éléments `1`, `2`, `4` et `7`.
- `[]` est la **liste vide**. Son type est `'a list`.
- `[[3; 4]; []; [9; 3; 4; 9]; [1]]` est une liste de listes d'entiers. Son type est `int list list`.

Si `lst = [e_1; e_2; ..., e_n]` est une liste non vide,

- l'élément `e_1` est la **tête** de `lst` ;
- la liste `[e_2; ...; e_n]` est la **queue** de `lst` .

La fonction

```
List.hd : 'a list -> 'a
```

renvoie la tête d'une liste non vide.

Ce qui suit le `:` est le type de la fonction. Ceci sera détaillé plus tard, mais pour le moment, nous pouvons retenir que cela signifie que la fonction accepte une valeur de type `'a list` et renvoie une valeur de `'a` .

La fonction

```
List.tl : 'a list -> 'a list
```

renvoie la queue d'une liste non vide.

L'**opérateur de construction** est l'opérateur infixe `::`.

Il prend un élément `x` et une liste `lst` et renvoie la liste dont la tête est `x` et la queue est `lst`.

Cet opérateur est un **raccourci infixe** pour la fonction

```
List.cons : 'a -> 'a list -> 'a list
```

Exemples

- `43 :: [1; 4; 9]` est la liste `[43; 1; 4; 9]`.
- `1 :: 2 :: [3; 4; 5]` est la liste `[1; 2; 3; 4; 5]`.
- `List.cons 43 [1; 4; 9]` est la liste `[43; 1; 4; 9]`.
- `List.cons 1 (List.cons 2 [3; 4; 5])` est la liste `[1; 2; 3; 4; 5]`.

Il est possible de se servir de `::` dans un motif.

Cet opérateur joue dans ce cas un rôle de **déconstructeur**.

Exemples

```
let is_empty lst =  
  match lst with  
  | x1 :: lst' -> false  
  | lst' -> true
```

Cette fonction teste si la liste `lst` est vide.

```
let two_or_more lst =  
  match lst with  
  | x1 :: x2 :: lst' -> true  
  | lst' -> false
```

Cette fonction teste si la liste `lst` possède au moins deux éléments.

Voici quelques exemples de fonctions sur les listes.

Exemple

```
let rec last lst =  
  match lst with  
  | [x] -> x  
  | x :: lst' -> last lst'
```

Cette fonction renvoie le dernier élément de la liste non vide `lst`.

```
let rec is_length_even lst =  
  match lst with  
  | [] -> true  
  | x :: lst' -> not (is_length_even lst')
```

Cette fonction teste si la liste `lst` est de longueur paire.

```
let rec member x lst =  
  match lst with  
  | [] -> false  
  | x' :: lst' ->  
    if x' = x then true else member x lst'
```

Cette fonction teste si la liste `lst` contient l'élément `x`.

2. Notions générales

2.1. Bases théoriques 59

2.2. Caractéristiques principales des langages de programmation 89

2.1. Bases théoriques

2.1.1. Brève chronologie de la programmation 61

2.1.2. Machines de Turing 65

2.1.3. Programme = entier 74

2.1.4. Décidabilité et indécidabilité 78

2.1.5. λ -calcul 83

2.1.1. Brève chronologie de la programmation

Préhistoire.

- 1801 : J. M. Jackard met au point la 1re machine programmable, un **métier à tisser**.
- 1837 : C. Babbage propose les plans de la **machine analytique**.
- 1843 : A. Lovelace propose le tout premier **programme**, destiné à la machine analytique.
- 1920 : M. Schönfinkel introduit la **logique combinatoire**.
- 1936 : A. Church introduit le **λ -calcul**.
- 1936 : A. Turing invente la **machine de Turing**.

Histoire.

- Années 1940 : **débuts de la programmation**, en assembleur et en langage machine.
- Années 1950-1960 : premiers vrais **langages de programmation** : FORTRAN (1957), Lisp (1958) et COBOL (1959).
- Années 1960-1970 : apparition des **paradigmes de programmation** : impératif, fonctionnel, orienté objet, logique.

Période moderne.

- Années 1970-2000 : apparition des **langages de programmation modernes** : Pascal (1970), C (1972), ML (1973), Prolog (1973), C++ (1983), (O)Caml (1987, 1996), Python (1991) et Java (1995).
- Années 1970-2000 : apparition d'**autres paradigmes de programmation** et de **méthodes de développement** : programmation par réécriture, programmation réactive, programmation concurrente, programmation extrême, développement piloté par les tests.
- Années 1990-2020 : essor des **bibliothèques** : NumPy (1995), scikit-learn (2007), pandas (2008), PyTorch (2016).
- Années 2022-2026 : avènement des **outils génératifs** pour l'assistance au développement.

2.1.2. 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. $\Delta : E \times \{., 0, 1\} \rightarrow E \times \{., 0, 1\} \times \{G, D\}$ est une fonction de transition.

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** :

u_1	u_2	$\dots$	$u_{ u }$	.	.	$\dots$
-------	-------	---------	-----------	---	---	---------

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

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

u_1	u_2	$\dots$	$u_{ u }$	.	.	$\dots$
-------	-------	---------	-----------	---	---	---------

Elle pointe vers la case surlignée. On appelle a la lettre dans $\{., 0, 1\}$ 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**.

Pour réaliser les actions dictées par le programme de M , on procède itérativement 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 $s' = D$, 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 à la première étape.

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

Le reste du ruban à droite est donc composé d'une infinité de $.$.

Exemple

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

$(e_1, 0) \rightarrow (e_1, 1, D)$

$(e_1, 1) \rightarrow (e_1, 0, D)$

$(e_1, \cdot) \rightarrow (e_2, \cdot, D)$

Registre d'état	Ruban						
e_1	<table><tr><td>0</td><td>0</td><td>1</td><td>.</td><td>.</td><td>...</td></tr></table>	0	0	1	.	.	...
0	0	1	.	.	...		
e_1	<table><tr><td>1</td><td>0</td><td>1</td><td>.</td><td>.</td><td>...</td></tr></table>	1	0	1	.	.	...
1	0	1	.	.	...		
e_1	<table><tr><td>1</td><td>1</td><td>1</td><td>.</td><td>.</td><td>...</td></tr></table>	1	1	1	.	.	...
1	1	1	.	.	...		
e_1	<table><tr><td>1</td><td>1</td><td>0</td><td>.</td><td>.</td><td>...</td></tr></table>	1	1	0	.	.	...
1	1	0	.	.	...		
e_2	<table><tr><td>1</td><td>1</td><td>0</td><td>.</td><td>.</td><td>...</td></tr></table>	1	1	0	.	.	...
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.

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

$(e_1, 0) \rightarrow (e_1, 0, D)$

$(e_1, 1) \rightarrow (e_2, 1, G)$

$(e_1, \cdot) \rightarrow (e_1, \cdot, D)$

Registre d'état	Ruban						
e_1	<table><tr><td>0</td><td>0</td><td>1</td><td>.</td><td>.</td><td>...</td></tr></table>	0	0	1	.	.	...
0	0	1	.	.	...		
e_1	<table><tr><td>0</td><td>0</td><td>1</td><td>.</td><td>.</td><td>...</td></tr></table>	0	0	1	.	.	...
0	0	1	.	.	...		
e_1	<table><tr><td>0</td><td>0</td><td>1</td><td>.</td><td>.</td><td>...</td></tr></table>	0	0	1	.	.	...
0	0	1	.	.	...		
e_2	<table><tr><td>0</td><td>0</td><td>1</td><td>.</td><td>.</td><td>...</td></tr></table>	0	0	1	.	.	...
0	0	1	.	.	...		

Lors du calcul de $M(001)$, nous arrivons à e_2 qui est l'état terminal : l'exécution est terminée.

Registre d'état	Ruban						
e_1	<table><tr><td>0</td><td>0</td><td>0</td><td>.</td><td>.</td><td>...</td></tr></table>	0	0	0	.	.	...
0	0	0	.	.	...		
e_1	<table><tr><td>0</td><td>0</td><td>0</td><td>.</td><td>.</td><td>...</td></tr></table>	0	0	0	.	.	...
0	0	0	.	.	...		
e_1	<table><tr><td>0</td><td>0</td><td>0</td><td>.</td><td>.</td><td>...</td></tr></table>	0	0	0	.	.	...
0	0	0	.	.	...		
e_1	<table><tr><td>0</td><td>0</td><td>0</td><td>.</td><td>.</td><td>...</td></tr></table>	0	0	0	.	.	...
0	0	0	.	.	...		
e_1	<table><tr><td>0</td><td>0</td><td>0</td><td>.</td><td>.</td><td>...</td></tr></table>	0	0	0	.	.	...
0	0	0	.	.	...		

Lors du calcul de $M(000)$, la tête de lecture/écriture part vers l'infini : l'exécution ne se termine pas.

Un langage de programmation $\mathcal{L}$ est **Turing-complet** s'il permet de simuler une machine de Turing.

Plus précisément, c'est le cas lorsqu'il est possible de développer dans $\mathcal{L}$ une **fonction d'émulation**

- paramétrée par le codage d'une machine de Turing M et le codage d'un mot $u \in \{0,1\}^*$
- qui renvoie le codage du résultat $M(u)$.

Ceci est un exercice très facile pour des **langages impératifs** comme C, Python ou Java.

Ils sont donc Turing-complets.

Le **ruban** implanté ne peut pas être infini mais est considéré comme **arbitrairement grand**.

Exemple

En Python, une telle fonction pourrait avoir l'allure suivante.

```
def emul(machine_encoding, input_encoding):  
    res = "" # A string of 0s and 1s.  
    ...  
    # Modification of res according with  
    # machine_encoding and input_encoding.  
    ...  
    return res
```

Note : il existe des langages intéressants qui ne sont pas Turing-complets (comme les langages rationnels, les langages algébriques et certains langages d'assistants de preuve).

L'absence de la propriété de Turing-complétude n'est pas obligatoirement un défaut : elle rend au contraire certaines propriétés sur leurs programmes démontrables (comme l'absence de boucles infinies par exemple).

Une **machine de Turing universelle** est une machine de Turing M_{univ} telle que, pour toute machine de Turing M et tout mot $u \in \{0,1\}^*$,

$$M_{\text{univ}}(v) = M(u),$$

où $v \in \{0,1\}^*$ est le codage du couple (M, u) .

En d'autres termes, M_{univ} est une machine de Turing qui permet d'**émuler n'importe quelle machine de Turing**.

Étant donné un langage de programmation $\mathcal{L}$, il est assez facile de traduire les constructions qu'il propose (affectations, instructions de branchement, *etc.*) de sorte à pouvoir construire, pour tout programme P en $\mathcal{L}$, une machine de Turing M_P qui émule l'exécution de P .

La **machine de Turing universelle** M_{univ} peut ainsi émuler M_P et donc, par transitivité, aussi P . Ainsi, M_{univ} peut calculer tout ce qu'un programme peut calculer.

2.1.3. Programme = entier

Nous associons à toute machine de Turing M un entier naturel $\text{code}(M)$ défini de la manière suivante :

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

Exemple

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

$$(e_1, 0) \rightarrow (e_1, 0, D)$$

$$(e_1, 1) \rightarrow (e_2, 1, G)$$

$$(e_1, \cdot) \rightarrow (e_1, \cdot, D)$$

Cette machine de Turing 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)
```

Nous en déduisons la représentation binaire

$$v = 01100101\ 01110100\dots00101001$$

et de celle-ci, l'entier naturel $\text{code}(M)$.

En décimal, ce nombre est

```
1195273483522463947698188428566573994862939056290801762903598135757140294873881005500260610437287957403372
26979189145773724736737165168419101329132422751418637824051118841640585439990747361814064299553649044252
2997516654502946689283241263412326824166734535399938860445815033689444918572055862472186480808828981756969
```

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.

Nous pouvons ainsi ordonner l'ensemble de toutes les machines de Turing : on pose $M \leq M'$ si $\text{code}(M) \leq \text{code}(M')$.

L'ensemble de toutes les machines de Turing peut donc être ordonné en un segment infini

$$M_0 < M_1 < M_2 < \dots < M_n < \dots$$

Le **rang** $\text{rang}(M)$ d'une machine de Turing M est la position de M dans ce segment, c.-à-d. $\text{rang}(M_n) = n$.

L'application **rang** fournit ainsi 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.

2.1.4. Décidabilité et indécidabilité

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

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 de programmation Turing-complet, une fonction f paramétrée par un objet u et renvoyant `true` si $P(u) = \text{oui}$ et `false` sinon.

Le **problème de l'arrêt** est le problème de décision **Arr** prenant en entrée le codage binaire d'un programme f (une fonction) et le codage binaire d'une entrée a (un argument) et renvoyant **oui** si l'exécution du calcul $f(a)$ se termine et **non** sinon.

Théorème – Turing (1936)
Le problème de l'arrêt Arr est indécidable .

Intuitivement, ceci dit que pour tout langage de programmation L , il est impossible de concevoir une fonction **halt** en L qui accepte en entrée une autre fonction f et un argument a , et qui teste si l'exécution de $f(a)$ se termine.

Montrons le résultat précédent **par l'absurde** en supposant que `halt` existe.

En pseudo-code (qui est bien un langage Turing-complet), cette fonction s'exprime par

```
fonction halt(f, a):  
    si f(a) se termine:  
        renvoyer oui  
    sinon:  
        renvoyer non
```

Soit maintenant la fonction `delta` définie par

```
fonction delta(x):  
    si halt(x, x) = oui:  
        renvoyer delta(x)    (A)  
    sinon:  
        renvoyer oui        (B)
```

L'appel `delta(delta)` produit une absurdité :

- si `delta(delta)` se termine, par (A), `delta(delta)` ne se termine pas ;
- si `delta(delta)` ne se termine pas, par (B), `delta(delta)` se termine.

Nous faisons donc face à une absurdité qui montre que la fonction `halt` n'existe pas.

2.1.5. λ -calcul

Une **fonction récursive** est une fonction

$$f : \mathbb{N}^k \rightarrow \mathbb{N}, \quad k \geq 0,$$

qui est intuitivement calculable.

Exemples

- $f : (n_1, n_2, n_3) \rightarrow n_1 + n_2 + n_3$;

Ces fonctions sont récursives au sens précédent.

- $f : n \rightarrow \begin{cases} 1 & \text{si } n \text{ est pair} \\ 0 & \text{sinon} \end{cases}$;

La dernière est même récursive au sens usuel.

- $f : n \rightarrow \begin{cases} 1 & \text{si } n \leq 1 \\ n \times f(n-1) & \text{sinon} \end{cases}$.

Exemple

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

$$f(n) := \begin{cases} 1 & \text{si Arr}(g) = \text{oui où } g \text{ est le programme tel que rang}(g) = n \\ 0 & \text{sinon} \end{cases}$$

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

Le λ -calcul a été introduit (entre autres) dans le but de proposer un formalisme pour définir et décrire les fonctions calculables.

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

Une expression du λ -calcul (ou λ -terme) est

1. soit une variable, notée x , y , z , ... ;
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.

Exemples

Quelques expressions :

- x
- $x x$
- $\lambda x.x$
- $\lambda x.\lambda y.x$
- $z (\lambda x.\lambda y.x)$
- $(\lambda z.z) ((\lambda x.x) (\lambda y.((\lambda x.x) y)))$

Une occurrence d'une variable x est **liée** si elle fait partie d'une sous-expression qui est dans la portée d'une abstraction en x .

Une occurrence d'une variable qui n'est pas liée est **libre**.

Une expression qui ne contient aucune variable libre est **close**.

Exemples

Les occurrences de variables qui sont **soulignées** sont **liées**. Les autres sont libres.

1. $\lambda x.x$

2. $\lambda x.y$

3. $\lambda x.\lambda y.(y x)$

4. $x (\lambda x.y (\lambda y.(x y)))$

5. $(\lambda x.y x) (\lambda y.y x)$

6. $\lambda z.(\lambda x.(y x) z) (\lambda y.(y x) z)$

La 1re et la 3e expression ici sont closes. Aucune autre ne l'est.

La β -substitution en racine consiste, étant donnée une expression de la forme $(\lambda x.f) a$, à substituer a aux occurrences libres de x dans f . Ceci forme l'expression $f[x := a]$. On note

$$(\lambda x.f) a \rightarrow f[x := a].$$

Exemple

$$(\lambda x.(x y) x) (z z) \rightarrow ((z z) y) (z z)$$

Une β -substitution d'une expression e_1 est une expression e_2 obtenue en remplaçant une sous-expression e'_1 de e_1 par l'expression e'_2 obtenue par β -substitution en racine de e'_1 . Par extension, on note $e_1 \rightarrow e_2$.

Exemple

$$x ((\lambda x.(x y) x) (z z)) \rightarrow x (((z z) y) (z z))$$

Attention : il y a ici une difficulté cachée qui survient lorsque a possède des variables libres qui se font capturer après une β -substitution.

Une **valeur** est une expression sur laquelle il n'est pas possible d'appliquer de β -substitution.

Exemples

- x est une valeur ;
- $\lambda x.x y$ est une valeur.
- $(\lambda x.x) y$ n'est pas une valeur ;
- $\lambda x.(x (\lambda y.\lambda z.z (y x))) z$ n'est pas une valeur.

L'**évaluation** d'une expression e est l'expression obtenue en appliquant des β -substitutions sur e jusqu'à obtenir une valeur. Elle peut ne pas exister.

Exemple

Voici une telle suite de β -substitutions jusqu'à l'obtention d'une valeur :

$$((\lambda x.\lambda y.x) u) v \rightarrow (\lambda y.u) v \rightarrow u$$

Il existe des expressions qui **n'aboutissent pas à des valeurs**, comme $(\lambda x.(x x)) (\lambda x.(x x))$.

2.2. Caractéristiques principales des langages de programmation

2.2.1. Paradigmes 91

2.2.2. Vérification des types 97

2.2.3. Attribution des types 102

2.2.4. Portée d'un identificateur 108

2.2.5. Pureté fonctionnelle 113

2.2.6. Caractéristiques de langages usuels 115

2.2.1. Paradigmes

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.

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

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

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

Un programme est une expression, celle-ci étant une imbrication de définitions et d'appels de fonctions.

L'exécution d'un programme est l'évaluation de son expression en utilisant les définitions des fonctions pour la simplifier au maximum.

Les éléments suivants sont constitutifs de ce paradigme :

1. liaison d'un nom à une valeur ;
2. récursivité ;
3. instructions de branchement ;
4. 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. Toutes les données sont représentées dans l'expression en cours d'évaluation.

Dans ce cours, nous adopterons au maximum le paradigme fonctionnel.

Ainsi, nous nous interdirons

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

En revanche, nous utiliserons de manière courante

1. des fonctions récursives ;
2. des fonctions locales.

Note : une constante peut-être vue comme une fonction d'arité zéro (c.-à-d. une fonction qui ne prend pas d'entrée).

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 pure, le principe de transparence référentielle s'applique.

2.2.2. Vérification des types

Dans la plupart des langages de programmation, les expressions sont **classifiées** selon la nature de la valeur qu'elles dénotent. On parle alors de **type**.

Ceci permet, lorsque des fonctions sont associées (par le biais par exemple de l'application d'une fonction à des arguments), de s'assurer qu'il n'y a pas d'incohérence manifeste. L'avantage est de pouvoir **capturer des erreurs** le plus tôt possible.

Exemple
Dans un langage typé proposant une fonction <code>add</code> acceptant deux entiers et renvoyant un entier, l'expression <code>add(10, "3")</code> serait mal typée car le 2e argument n'est pas un entier.

Il existe plusieurs stratégies de **vérification des types** :

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 ;
3. **vérification hybride**, où certains aspects sont vérifiés lors de la compilation et d'autres, lors de l'exécution du programme.

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 dans la vérification des types peuvent se produire :

- si l'utilisateur saisit un entier, alors 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__'
```

Exemple

Considérons le programme C, similaire au précédent,

```
#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 1.8 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]);
                        ^
```

Vérification des types dynamique.

- Avantage : grande flexibilité dans l'écriture des programmes. L'ensemble des expressions correctes est plus large.
- Inconvénients : les problèmes de types ne sont mis en évidence qu'à l'exécution. Le programmeur doit donc faire attention à tester tous les cas de figure. Il y a une perte d'efficacité lors de l'exécution à cause de la vérification des types qui se fait au même moment et qui demande des ressources.

Vérification des types statique.

- Avantages : grande sécurité lors de l'écriture du programme. Les erreurs les plus courantes sont détectées à la compilation. L'exécution n'est pas accompagnée de la vérification des types, ce qui peut la rendre plus rapide.
- Inconvénient : moins de flexibilité dans l'écriture des programmes. L'ensemble des expressions correctes est plus restreint.

2.2.3. Attribution des types

Dans un programme dans un langage donné, il y a deux manières principales d'**attribuer un type** à une expression ou à une sous-expression donnée.

1. L'**attribution explicite** (nommée parfois **Church-style**) demande que les **types** des variables et fonctions sont **mentionnés** dans le programme.

Dans ce cas, la vérification des types s'appuie sur les **annotations de types** mentionnées dans le programme afin de voir si elles sont cohérentes.

2. L'**attribution implicite** (nommée parfois **Curry-style**) demande que 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**.

3. L'**attribution hybride**, une approche hybride, qui demande **certaines annotations de types** (pour les expressions atomiques notamment) et infère le type des expressions composées.

Exemple

Considérons le programme OCaml

```
let pair (x : int) (y : bool) : (int * bool) =  
  (x, y)  
  
let () =  
  let p = pair 2 true in  
  print_int (fst p)
```

Les **annotations de types explicites** indiquent que `pair` est une fonction acceptant un entier et un booléen et renvoyant un couple dont la 1re composante est un entier et la 2e est un booléen.

Ces annotations de types forment aussi des **contraintes** : par exemple, l'appel `pair 2 5` provoquerait une erreur lors de la vérification des types.

Exemple

```
let pair x y =  
  (x, y)  
  
let () =  
  let p1 = pair 2 true in  
  print_int (fst p1);  
  let p2 = pair 2 5 in  
  print_int (snd p2)
```

Considérons le programme OCaml ci-contre, similaire au précédent.

Ici, les **annotations de types** sont **implicites** au niveau de la définition de la fonction `pair`.

Le système d'inférence des types lui attribue automatiquement un type. Il devine le **type le plus général possible** pour cette fonction.

Ce type est `'a -> 'b -> 'a * 'b`, mentionnant que `pair` est une fonction acceptant une valeur de n'importe quel type (`'a`), une valeur de n'importe quel type (`'b`, potentiellement différent de `'a`) et renvoyant un couple dont la 1re composante est de type `'a` et la 2e est de type `'b`.

Nous observons ici que la fonction `pair` est appelée deux fois et avec des types d'arguments différents.

Tout ceci est en lien avec le **polymorphisme** (notion qui sera vue en détail plus tard).

Exemple

Considérons le programme OCaml

```
let next x =  
  x + 1  
  
let () =  
  print_int (next 5)
```

Ici, `next` 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 système d'inférence des types suggère que `next` 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.

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 système d'inférence des types se charge de trouver ceux qui sont les plus adaptés et les plus généraux.
- Inconvénients : programmes moins lisibles. Si la vérification des types est statique, la compilation peut être plus longue (il faut deviner les types). Si la vérification des types est dynamique, l'exécution peut être moins efficace.

Note : dans certains langages, il est possible d'attribuer explicitement certains types et de laisser le système d'inférence de types en deviner certains. L'OCaml s'inscrit dans cette approche hybride.

2.2.4. Portée d'un identificateur

La **portée** d'un identificateur (de variable, de fonction, ou encore dans certains cas, de types) dans un programme désigne l'étendue dans laquelle cet identificateur existe.

Il existe deux principales sortes de portées :

1. **portée statique**, où la portée d'un identificateur dépend de sa **position** dans le programme ;
2. **portée dynamique**, où la portée d'un identificateur dépend de la façon dont le programme s'**exécute**. Elle peut donc varier d'une exécution à une autre.

Note : les identificateurs à portée dynamique sont peu courants dans les langages modernes.

Exemple

Considérons le programme OCaml

```
let () =  
  let x = 10 in  
  let y = x + 1 in  
  print_int (x + y)
```

Ici, la portée de `x` va de la l. 3 à la l. 4. La portée de `y` va de la l. 4 à la l. 4.

Ces portées sont **statiques** : elles ne dépendent que des positions des `let` définissant `x` et `y`.

Exemple

Considérons le programme OCaml

```
let () =  
  let x =  
    if read_int () mod 2 = 0 then 10 else 20  
  in  
  print_int x
```

Même idée ici : la portée de `x` va de la l. 5 à la l. 5 et est statique. Elle ne dépend pas de l'entier saisi sur l'entrée standard.

Exemple

Considérons le programme OCaml

```
let () =  
  let x = 10 in  
  let y1 = x + 1 in  
  
  let x = 20 in  
  let y2 = x + 1 in  
  
  print_int y1;  
  print_newline ();  
  print_int y2
```

Ici, il y a deux définitions de l'identificateur `x`. La portée du 1er s'étend de la l. 3 à la l. 4. La portée 2e s'étend de la l. 6 à la l. 10.

Le 1er identificateur `x` est **masqué** car un autre identificateur du même nom est déclarée dans la portée qu'il devait avoir initialement.

Ce phénomène s'appelle l'**ombrage de nom**.

Exemple

Considérons les programmes similaires OCaml et Bash

```
let x = 5

let f1 () =
  print_int x

let f2 () =
  let x = 10 in
  f1 ()

let () =
  f1 ();
  f2 ()
```

```
#!/bin/bash

x=5
function f1 {
  echo $x
}
function f2 {
  local x=10
  f1
}
f1
f2
```

Sans surprise, l'exécution du programme de gauche affiche 5 puis 5. En effet, la définition du `x` dans `f2` n'a pas d'influence sur le `x` de `f1` qui est dans la portée du `x` de la l. 1.

En revanche, l'exécution du programme de droite affiche 5 puis 10. En effet, le `x` défini en l. 9 à une portée incluant le code de `f1`. La portée de ce `x` est *dynamique*.

2.2.5. Pureté fonctionnelle

Il y a plusieurs niveaux de pureté dans les langages fonctionnels.

1. Dans un langage **fonctionnel pur**, la notion d'effet secondaire est inexistante.

Des difficultés se posent alors notamment lors de la gestion des entrées/sorties. Elles sont résolues élégamment au moyen des **monades**.

2. Dans un langage **fonctionnel impur**, certaines particularités des langages impératifs sont intégrées.

Cela englobe la gestion classique des entrées/sorties ou le fait d'autoriser certaines données à être mutables.

L'OCaml s'inscrit dans cette dernière catégorie. Nous ferons en effet appel de temps en temps à des fonctions d'impression.

2.2.6. Caractéristiques de langages usuels

Langage	Vérif. dyn.	Vérif. stat.	Attr. expl.	Attr. impl.	Impératif	Fonctionnel
C	Non	Oui	Oui	Non	Oui	Non
Python	Oui	Non	Non	Oui	Oui	Impur
Java	Non	Oui	Oui	Non	Oui	Non
OCaml	Non	Oui	Oui	Oui	Oui	Impur
Haskell	Non	Oui	Oui	Oui	Non	Pur
Rust	Non	Oui	Oui	Oui	Oui	Impur
Go	Non	Oui	Oui	Oui	Oui	Non
Scala	Non	Oui	Oui	Oui	Oui	Impur
Scheme	Oui	Non	Non	Oui	Oui	Impur
Common Lisp	Oui	Non	Non	Oui	Oui	Impur
PureScript	Non	Oui	Oui	Oui	Non	Pur
Prolog	Oui	Non	Non	Oui	Non	Non

3. Programmation fonctionnelle

- 3.1. Expressions de base 119
- 3.2. Fonctions 164
- 3.3. Entrées, sorties et séquences 199
- 3.4. Écriture de projets 209
- 3.5. Types 222
- 3.6. Récursivité 266
- 3.7. Filtrage de motifs 287
- 3.8. Fonctions d'ordre supérieur 312
- 3.9. Polymorphisme 343
- 3.10. Listes 358
- 3.11. Stratégies d'évaluation 388
- 3.12. Non mutabilité 401
- 3.13. Preuves 416

3.1. Expressions de base

- 3.1.1. Généralités 121
- 3.1.2. Interpréteur 125
- 3.1.3. Liaisons 129
- 3.1.4. Types de base 141
- 3.1.5. Expression conditionnelles 157

3.1.1. Généralités

Le système d'exploitation conseillé est **Linux** (Manjaro, Debian, *etc.*).

Nous aurons besoin (au minimum) des logiciels et outils suivants :

- un **émulateur de terminal** (xterm, rxvt, alacritty, kitty, *etc.*) ;
- un **éditeur de texte** (emacs, vim, *etc.*) ;
- une **distribution OCaml** (opam fortement conseillé).

Le site officiel du langage

<https://ocaml.org>

regorge d'informations (pour l'installation, mais aussi pour de la documentation).

Le site

<https://ocaml.org/manual/5.5/api/index.html>

contient la documentation de la **bibliothèque standard**.

Voir aussi

<https://github.com/ocaml-community/awesome-ocaml>

pour beaucoup de **ressources à propos du langage**.

Le langage OCaml est utilisé dans certains **projets d'envergure** comme

- MirageOS (<https://mirage.io>), une bibliothèque pour systèmes d'exploitation ;
- Flow (<https://flow.org>), un vérificateur de types statique pour JavaScript ;
- Hack (<https://hacklang.org>), un compilateur ;
- Liquidsoap (<https://www.liquidsoap.info>), un langage pour le streaming multimédia ;
- google-drive-ocamlfuse (<https://astrada.github.io/google-drive-ocamlfuse>), un système de fichiers FUSE pour Google Drive.

Il est utilisé dans plusieurs **entreprises importantes** comme

- Meta (<https://about.meta.com>), une des principales GAFAM ;
- Jane Street (<https://www.janestreet.com>), une entreprise de services financiers et d'opérations boursières quantitatives ;
- Tezos (<https://tezos.com>), une blockchain.

Voir <https://ocaml.org/industrial-users> pour d'autres informations.

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

Exemple

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

Les symboles `(*` et `*)` fonctionnent comme des parenthèses : il doit être possible d'appareiller chaque `(*` avec un unique `*)`.

Les **commentaires imbriqués** sont autorisés.

Exemples

En OCaml, ceci fonctionne :

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

En C, ceci ne fonctionne pas :

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

Il n'existe pas de moyen de commenter une seule ligne en OCaml.

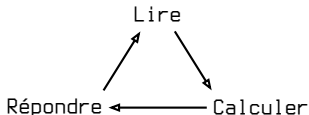
3.1.2. Interpréteur

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

```
Objective Caml version 5.5.0
...
#
```

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.



Ce cycle est suivi jusqu'à ce que l'utilisateur l'interrompe en saisissant CTRL+D, `exit 0;;`, ou encore `#quit;;`.

Il existe un **interpréteur amélioré** : `utop`.

Cet interpréteur propose diverses fonctionnalités pratiques comme

- une gestion de l'**historique** (touches HAUT et BAS) ;
- un accès à l'**autosuggestion des noms** (alias, fonctions, types, *etc.*). Il apparaissent en dessous de la ligne d'entrée ;
- une gestion de l'**autocomplétion des noms**. Il est possible de sélectionner un nom suggéré par ALT+GAUCHE et ALT+DROITE et de l'utiliser par ALT+TAB.

Il est possible de **charger un fichier** `File.ml` dans `utop` (ou dans `ocaml`) par

```
#use "File.ml";;
```

Le symbole `#` fait partie de la commande.

Ceci permet d'avoir accès, dans l'interpréteur, aux entités définies dans `File.ml`.

Une **phrase** est une **expression** ou une **déclaration** terminée par `::` (fin de phrase).

Exemple

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

Une phrase peut tenir sur plusieurs lignes. Les retours à la ligne n'ont pas d'influence sur son sens.

Exemple

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

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

Exemple

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

Explications :

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

3.1.3. Liaisons

De la même manière que nous pouvons écrire

« Soit *n* l'entier 5. »

pour définir ce que représente le symbole « *n* » pour avoir le droit de l'utiliser ensuite, il est possible en programmation de donner un nom à une valeur.

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

On utilise pour cela la construction syntaxique

let ID = EXP

où **ID** est un **nom** (identificateur) et **EXP** est une expression. Il s'agit ici d'une **liaison globale**.

Exemple

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

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

La 2e phrase donne la valeur à laquelle *n* est lié.

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

La **portée** d'un nom défini localement est étendue à l'expression où figure sa définition. Nous utilisons pour cela la construction syntaxique

let ID = EXP1 in EXP2

où **ID** est un nom et **EXP1** et **EXP2** sont des expressions.

C'est une **liaison locale**.

Très important : cette expression **possède comme valeur** celle de **EXP2** dans laquelle les occurrences libres de **ID** sont remplacées par la valeur de **EXP1**.

Voici quelques exemples simples de liaisons locales.

Exemple

```
# 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`, ainsi donc que toute l'expression.

Exemple

```
# 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.

Ce nom local n'existe plus en dehors, ce qui fait que dans la 3e phrase, le `n` global est considéré.

Pour comprendre ce à quoi fait référence un nom, il est important de visualiser la portée de chaque identificateur défini par une liaison.

Il est alors possible **relier** chaque **occurrence** d'un nom à sa **définition**.

Il est très important de tenir compte du phénomène d'**ombrage de nom** dans le cas où un nom **ID** est lié au sein de la portée d'un nom identique **ID**.

Voici quelques exemples utilisant des liaisons locales et globales avec des phénomènes d'ombrage de nom.

Exemple

```
# 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 imprimé.

Exemple

```
# 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`.

Il est possible de mettre cette sous-expression entre parenthèses pour améliorer la lisibilité.

Voici d'autres exemples utilisant des liaisons locales et globales avec des phénomènes d'ombrage de nom.

Exemple

```
# 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 1. 3 n'est pas utilisé pour attribuer une valeur au nom `x`.

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

Exemple

```
# 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 1. 2 n'est pas défini.

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 = EXP1 and ID2 = EXP2
```

ou

```
let ID1 = EXP1 and ID2 = EXP2 in EXP3
```

où **ID1** et **ID2** sont des identificateurs, et **EXP1**, **EXP2** et **EXP3** sont des expressions.

Cette construction est particulièrement utile pour définir des noms **mutuellement récursifs** (cas exploré plus loin).

Voici quelques exemples de base sur les liaisons simultanées.

Exemple

```
# 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`.

Exemple

```
# 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.

Notons l'usage particulier de l'indentation impliqué par les `in` et les `and`.

Voici quelques exemples qui illustrent quelques particularités des liaisons simultanées.

Exemple

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

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

Exemple

```
# 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 dans la phrase précédente le `and` par un `in let`.
Le résultat est différent du précédent : dans la définition locale du nom `y`, l'occurrence de `x` qui `y` apparaît est celle définie en l. 2.

Voici encore deux autres exemples supplémentaires.

Exemple

```
# 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.

Les occurrences du nom `x` en l. 4 font référence à la définition de `x` en l. 3.

Cette définition n'influe pas sur la définition de `x` en l. 1.

Exemple

```
# 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.

Les liaisons `let ID = EXP` et les liaisons locales `let ID = EXP1 in EXP2` sont des entités totalement différentes.

En effet,

- une liaison globale `let ID = EXP` est une phrase qui **n'a pas de valeur**.

Son but est de **déclarer** un alias `ID` pour la valeur de `EXP` qui sera **visible globalement** dans la suite du programme.

Ceci **n'est pas une expression** mais une **déclaration**.

- Une liaison locale `let ID = EXP1 in EXP2` est une phrase qui **a une valeur** et qui est celle de `EXP2` dans laquelle les occurrences de `ID` sont remplacées par la valeur de `EXP1`.

Ceci **n'est pas une déclaration** mais une **expression**.

Exemple

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

Exemple

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

3.1.4. Types de base

Nom du type	Utilisation
<code>int</code>	Représentation des entiers signés
<code>float</code>	Représentation des nombres à virgule signés
<code>char</code>	Représentation des caractères
<code>string</code>	Représentation des chaînes de caractères
<code>bool</code>	Représentation des booléens
<code>unit</code>	Type contenant une unique valeur

Le type `bool` contient exactement deux valeurs : `true` et `false`.

Voici les **opérations booléennes** :

Opérateur	Arité	Rôle
<code>not</code>	1	Non logique
<code>&&</code>	2	Et logique
<code> </code>	2	Ou logique

Ces opérateurs produisent des valeurs de type `bool`.

Exemples

```
# (false || (not false)) &&  
  (not (true || false));;  
- : bool = false
```

```
# not true && false;;  
- : bool = false
```

Règle : ne pas hésiter à introduire des parenthèses pour gagner en lisibilité.

Règle (encore meilleure) : apprendre les priorités usuelles entre les opérateurs pour limiter les parenthèses vraiment inutiles.

Une valeur de type `int` peut s'écrire en

- **décimal**, sans préfixe ;
- **hexadécimal**, avec le préfixe `0x` ;
- **binaire**, avec le préfixe `0b` .

Exemples
• <code>0</code> , <code>1024</code> , <code>-82</code> • <code>0x0</code> (entier <code>0</code>), <code>0x400</code> (entier <code>1024</code>), <code>-0xAE00F23</code> (entier <code>-182456099</code>) • <code>0b1011011</code> (entier <code>91</code>), <code>-0b101</code> (entier <code>-5</code>)

La **plage** des `int` s'étend de `min_int` à `max_int` .

Sur un système `64` bits, ceci va de

$$-2^{62} = -4611686018427387904 \text{ à } 2^{62} - 1 = 4611686018427387903.$$

La plage ne s'étend pas de -2^{n-1} à $2^{n-1} - 1$. Il y a effet utilisation d'un bit pour la gestion automatique de la mémoire.

Voici les **opérations arithmétiques** sur les `int` :

Opérateur	Arité	Rôle
<code>-</code> , <code>+</code>	1	Moins, Plus (signe)
<code>-</code> , <code>+</code>	2	Soustraction, Addition
<code>/</code> , <code>*</code>	2	Division, Multiplication
<code>succ</code>	1	Successeur
<code>pred</code>	1	Prédécesseur
<code>abs</code>	1	Valeur absolue
<code>mod</code>	2	Modulo

Ces opérateurs produisent des valeurs de type `int`.

Quelques remarques :

- `succ` , `pred` et `abs` sont des fonctions ;
- `mod` n'est pas une fonction, c'est un opérateur.

Voici les **opérations relationnelles** sur les `int` :

Opérateur	Arité	Rôle
<code>=</code> , <code><></code>	2	Égalité, Différence
<code><</code> , <code>></code>	2	Comparaison stricte
<code><=</code> , <code>>=</code>	2	Comparaison large

Ces opérateurs produisent des valeurs de type `bool` .

Exemples

```
# (2 = 1) || (32 <= 64);;  
- : bool = true
```

```
# let x = 2 and y = 3 in x = y;;  
- : bool = false
```

Important : ne jamais utiliser l'opérateur `==` qui teste l'égalité physique au lieu de l'égalité sémantique (chose que fait `=`).

L'opérateur `==` est intéressant dans un contexte où les données mutables existent, ainsi que la notion de référence, ce qui n'est pas notre cas.

Remarque : nous verrons que ces opérations fonctionnent aussi sur des valeurs ayant des types différents de `int` (et même sur des types qui ne sont pas de base).

Voici les **opérations bit à bit** sur les `int` :

Opérateur	Arité	Rôle
<code>lnot</code>	1	Non bit à bit
<code>land</code>	2	Et bit à bit
<code>lor</code>	2	Ou bit à bit
<code>lxor</code>	2	Ou exclusif bit à bit
<code>lsl</code> , <code>lsr</code>	2	Décalage à gauche, droite bit à bit
<code>asr</code>	2	Décalage à droite avec respect du signe bit à bit

Ces opérateurs produisent des valeurs de type `int` .

Exemples	
<pre># 1 lsl 10;; - : int = 1024</pre>	<pre># (lnot 0) lsr 1;; - : int = 4611686018427387903</pre>

Le type `float` permet de représenter des nombres à virgule.

Une valeur de type `float` s'écrit obligatoirement avec une virgule « `.` ».

Exemple
4.52 est l'écriture du nombre 4.52.

Le zéro à virgule s'écrit `0.0` ou encore `0.`.

Exemples	
<pre># 0;; - : int = 0</pre>	<pre># 0.;; - : float = 0.</pre>

La **plage** des `float` s'étend de `-max_float` à `max_float`.

Sur un système `64` bits, ceci va de

$$-\text{max_float} = -1.79769313486231571 \times 10^{308} \quad \text{à} \quad \text{max_float} = 1.79769313486231571 \times 10^{308}.$$

Règle : les opérations arithmétiques sur les `int` ont leur analogue sur les `float` en ajoutant un « `.` » à l'opérateur, ce qui donne `-.`, `+`, `/.`, `*.`.

Attention : on ne peut pas mélanger les `int` et les `float` de manière non explicite.

Il est possible de **convertir** un `int` en `float` par la fonction `float_of_int` et un `float` en `int` par la fonction `int_of_float` (réalisant une troncature).

Exemple

```
# 1. +. 1.;;
- : float = 2.
```

Exemple

```
# 2 +. 3.5;;
Error: This expression has type int but an
expression was expected of type float
```

Exemples

```
# float_of_int 32;;
- : float = 32.
```

```
# int_of_float 21.9;;
- : int = 21
```

Les opérateurs relationnels sur les `float` sont les mêmes que ceux sur les `int`.

Exemples

```
# 32. <= 89.99;;  
- : bool = true
```

```
# 2. *. 3. = 2. *. 2. +. 2. *. 2.;;  
- : bool = false
```

Observation : il peut paraître à première vue surprenant que les opérateurs relationnels soient communs aux `int` et aux `float` sans que les opérateurs arithmétiques le soient. Ce point sera élucidé dans la suite.

Ici aussi, il est impossible de mélanger les `int` et les `float` de manière non explicite.

Exemple

Il ne faut pas écrire

```
# 67.67 = 8;;  
Error: This expression has type int but an  
expression was expected of type float
```

mais plutôt

```
# 67.67 = (float_of_int 8);;  
- : bool = false
```

Il faut en effet demander explicitement la conversion d'un `int` en un `float` pour les utiliser au sein d'un opérateur (`=` ici).

Il existe des opérateurs spécifiques aux expressions de type `float`. Entre autres :

Opérateur	Arité	Rôle
<code>abs_float</code>	1	Valeur absolue
<code>floor</code> , <code>ceil</code>	1	Partie entière inf., sup.
<code>sqrt</code>	1	Racine carrée
<code>log</code> , <code>exp</code>	1	Logarithme népérien, exponentielle
<code>cos</code> , <code>sin</code> , <code>tan</code>	1	Fonctions trigonométriques
<code>**</code>	2	Exponentiation

Ces opérateurs produisent des valeurs de type `float`.

Hormis `**` qui est bien un opérateur du langage, les autres sont en réalité des fonctions prédéfinies.

Le type `char` permet de représenter les caractères ASCII.

Une valeur de type `char` peut s'écrire

- par un **caractère** entre apostrophes ;
- par son **code ASCII**, sur trois chiffres en base dix précédés de `\`, le tout entre apostrophes.

La fonction `int_of_char` calcule le code ASCII d'un caractère.

La fonction `char_of_int` calcule le caractère ayant le code ASCII spécifié.

Exemples

- `'a'`, `'8'`, `'?'`
- `'\101'`, `'\000'`, `'\035'`

Exemple

```
# int_of_char 'G';;  
- : int = 71
```

Exemples

```
# char_of_int 40;;  
- : char = '('
```

```
# char_of_int 2;;  
- : char = '\002'
```

Le type `string` permet de représenter des chaînes de caractères.

Une **chaîne de caractères** s'écrit par une suite de caractères entre guillemets.

Exemples

- `"abc123"`
- `"\100\101f"`

L'opérateur `^` permet de concaténer deux chaînes de caractères.

Plus précisément, si `u` et `v` sont des noms liés à des chaînes de caractères, `u ^ v` est une expression dont la valeur est la concaténation de `u` et de `v`.

Rappel important : il n'y a pas d'effet secondaire : les chaînes `u` et `v` ne sont **pas modifiées** lors de leur concaténation.

Exemples

```
# "abc" ^ "def";;  
- : string = "abcdef"
```

```
# let u = "ab" and v = "ba" in  
  u ^ v ^ u ^ v;;  
- : string = "abbaabba"
```

Il existe des **fonctions de conversion** autour du type `string`. Parmi celles-ci, il y a

- `string_of_bool` et `bool_of_string` ;
- `string_of_int` et `int_of_string` ;
- `string_of_float` et `float_of_string` .

Les **opérateurs relationnels** sont bien définis sur les valeurs de type `string`. On peut ainsi comparer des chaînes de caractères.

Les opérateurs de comparaison travaillent selon l'**ordre lexicographique**.

Exemples

```
# "abc" = "abde";  
- : bool = false
```

```
# "abc" <= "aaaa";  
- : bool = false
```

```
# let u = "ab" and v = "ab" in  
  u = v;;  
- : bool = true
```

```
# "abc" < "ad";  
- : bool = true
```

Le type `unit` est un type particulier qui contient une unique valeur, appelée « vide » et écrite

()

Il s'agit d'un **type singleton**.

Exemples

```
# ();;  
- : unit = ()
```

```
# let a = () in  
  let b = a in  
    b;;  
- : unit = ()
```

Les opérateurs relationnels sont bien définis sur `unit`.

Dans la suite, nous verrons que ce type sert à rendre les fonctions homogènes au sens où **toute fonction doit renvoyer une valeur**.

En effet, une fonction qui n'est pas censée renvoyer de valeur va renvoyer `()`.

Ce type sert également à introduire des **retardateurs** (notion qui sera abordée aussi dans la suite).

3.1.5. Expression conditionnelles

Une **expression conditionnelle** est une expression de la forme

```
if COND then EXP1 else EXP2
```

où **COND** est une expression de type **bool** et **EXP1** et **EXP2** sont deux expressions **toutes deux d'un même type**.

Exemples

```
# if (3 >= 2) || ("aab" <= "aa") then
  "ABC"
else
  "CDE";;
- : string = "ABC"
```

```
# let x =
  if 16. <= 2. ** 3. then 21 else 42;;
val x : int = 42
```

Toute expression conditionnelle **possède une valeur** :

1. lorsque **COND** s'évalue en **true**, l'expression conditionnelle a pour valeur celle de **EXP1** ;
2. lorsque **COND** s'évalue en **false**, l'expression conditionnelle a pour valeur celle de **EXP2** .

Une expression conditionnelle étant elle-même une expression, il est possible d'**imbriquer les expressions conditionnelles**.

Exemple

```
# if true then
  if 2 = 3 then
    'A'
  else
    'B'
else
  'C';;
- : char = 'B'
```

La (bonne) indentation aide à comprendre cette phrase.

Néanmoins, l'interpréteur OCaml la comprend sans ambiguïté.

Il n'a pas besoin de marqueur de fin (comme le `}` de certains langages).

Il est aussi possible d'utiliser des expressions conditionnelles sein d'autres opérations.

Exemple

```
# (if "ab" <= "baa" then 1 else 2)
+
  (if false then 3 else 2);;
- ; int = 3
```

Les valeurs des deux expressions conditionnelles sont additionnées.

L'expression est bien typée car les types des sous-expressions des conditionnelles sont identiques.

Rappelons qu'en programmation fonctionnelle, il n'y a pas d'effet secondaire.

Ainsi, l'intérêt d'une **expression conditionnelle** repose dans la **valeur** qu'elle exprime (et non pas dans l'action d'aiguiller l'exécution, propre au paradigme impératif).

Exemples

Voici quelques évaluations d'expressions conditionnelles :

- `if 3 >= 1 then 21 else 24` $\rightarrow$ `if true then 21 else 24` $\rightarrow$ 21 ,
- `if if "ab" = "ab" then 1 = 0 else true then 28 else 21`
 $\rightarrow$ `if if true then 1 = 0 else true then 28 else 21`
 $\rightarrow$ `if 1 = 0 then 28 else 21` $\rightarrow$ `if false then 28 else 21` $\rightarrow$ 21 .

Grâce au **principe de transparence référentielle**, ces deux expressions peuvent être remplacées par leurs valeurs, l'entier 21, dans tout programme les employant sans modifier la valeur finale qu'il calcule.

Une **demi-expression conditionnelle** est une expression de la forme

```
if COND then EXP
```

où `COND` est une expression de type `bool` et `EXP` est une expression.

Le système **complète implicitement** et automatiquement cette demi-expression conditionnelle en l'expression conditionnelle

```
if COND then EXP else ()
```

En conséquence, la valeur de `EXP` doit être de type `unit`.

Exemple

```
let f x =  
  if x >= 9 then ()
```

est équivalent à

```
let f x =  
  if x >= 9 then () else ()
```

Exemple

Voici un exemple plus concret :

```
# let x = read_int () in  
  if x <= 0 then print_int (-x);;
```

Ceci lit un entier et affiche sa valeur absolue s'il est négatif et ne fait rien sinon.

3.2. Fonctions

3.2.1. Rappels fondamentaux 167

3.2.2. Définitions et applications de fonctions 173

3.2.3. Types fonctionnels 178

3.2.4. Applications partielles et fonctions curryfiées 183

3.2.5. Fonctions anonymes 190

3.2.6. Inférence des types des fonctions 193

3.2.7. Retardateurs 196

Rappelons qu'en programmation fonctionnelle :

1. l'objet de base est la **fonction** ;
2. un programme est une **collection de définitions de fonctions** ;
3. un programme s'exécute en **évaluant sa fonction principale** ;
4. la définition d'une fonction sert à expliquer comment **calculer l'application de cette fonction à des arguments** ;
5. bien entendu, des **définitions de types** peuvent se trouver dans un programme (ce point sera abordé un peu plus loin).

La raison d'être d'un programme, en programmation fonctionnelle, est de **calculer une valeur**. Il s'agit de la **valeur de retour** calculée par la fonction principale.

Durant cette évaluation, dans un style fonctionnel non pur, des effets secondaires d'impression sur la sortie standard ou sur le système de fichiers ou encore de lecture d'une valeur sur l'entrée standard peuvent avoir lieu.

3.2.1. Rappels fondamentaux

Voici quelques rappels fondamentaux sur les ensembles, relations binaires et fonctions.

Un **ensemble** est une collection d'éléments.

Étant donné un ensemble E et une entité x , il y a deux possibilités mutuellement exclusives :

- $x \in E$. Nous disons alors que « x appartient à E » ou que « x est un élément de E ».
- $x \notin E$. Nous disons alors que « x n'appartient pas à E ».

Un ensemble E' est un **sous-ensemble** (également appelé **partie**) d'un ensemble E si pour toute entité x , $x \in E'$ implique $x \in E$. Cette propriété est notée $E' \subseteq E$.

Les **opérations ensemblistes** principales sont

- l'**union** $\cup$, telle que $x \in E_1 \cup E_2$ ssi $x \in E_1$ ou $x \in E_2$;
- l'**intersection** $\cap$, telle que $x \in E_1 \cap E_2$ ssi $x \in E_1$ et $x \in E_2$;
- le **produit cartésien** $\times$ tel que $x \in E_1 \times E_2$ ssi $x = (x_1, x_2)$ avec $x_1 \in E_1$ et $x_2 \in E_2$;
- l'**ensemble des parties** $\mathcal{P}$ tel que $\mathcal{P}(E)$ est l'ensemble des ensembles E' tels que $E' \subseteq E$.

Un **type** peut être vu comme un ensemble. Dire qu'une expression **EXP** est de type **T** revient à dire que **EXP** est un élément de l'ensemble **T**.

Attention : cette analogie a une limite. En effet, un élément peut appartenir à plusieurs ensembles (comme **0** qui est à la fois dans **N** et dans **R**), mais une valeur est, dans la plupart des systèmes de types, d'un seul type.

Une **relation binaire** entre deux ensembles E et E' est un élément de $\mathcal{P}(E \times E')$.

Ainsi, une relation binaire entre E et E' est un ensemble de couples (x, x') tels que $x \in E$ et $x' \in E'$.

Une **fonction** de **domaine** E et de **codomaine** E' est une relation binaire f entre E et E' telle que pour tout $x \in E$, il y a au plus un $x' \in E'$ tel que $(x, x') \in f$.

Cette propriété est notée

$$f : E \rightarrow E'.$$

Si pour un $x \in E$, il existe $x' \in E'$ tel que $(x, x') \in f$, x' est l'**image** de x . Nous notons alors par $f(x)$ cet élément x' .

Une fonction à n entrées est une fonction $f : (E_1 \times \dots \times E_n) \rightarrow E'$.

Nous notons alors par

$$f(x_1, \dots, x_n)$$

l'élément $f((x_1, \dots, x_n))$.

Remarque : lorsque $n = 0$, la fonction est de la forme $f : E'$. Il s'agit donc d'une constante.

Nous verrons un peu plus loin que les fonctions à plusieurs entrées sont conceptualisées de manière beaucoup plus ingénieuse et homogène en programmation fonctionnelle par la **curryfication**.

Il s'agit d'un concept très élégant et puissant.

Au sein de la manipulation des fonctions, il faut faire la distinction entre paramètre et argument :

1. un **paramètre** est un **identificateur** non lié à une valeur et intervenant dans la définition d'une fonction ;
2. un **argument** est une **expression** qui vient se substituer à un paramètre lors de l'appel à une fonction.

Exemple

Soit $f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ la fonction définie par $f(x_1, x_2) := x_1 + x_2$.

Les noms x_1 et x_2 sont les **paramètres** de f .

Lors de l'appel $f(f(2,1), 6-3)$, f est appelée avec les **arguments** $f(2,1)$ et $6-3$.

Lors d'un appel à une fonction f à n entrées avec les arguments $a_1, \dots, a_n$, nous disons que l'on **applique** f à $a_1, \dots, a_n$.

Exemple

Dans l'expression `max 2 (8 + 1)`, la fonction `max` est appliquée aux arguments `2` et `8 + 1`.

3.2.2. Définitions et applications de fonctions

Pour **définir une fonction** à n paramètres, nous utilisons la construction syntaxique

```
let ID P1 ... Pn = EXP
```

où **ID** est le nom de la fonction, **P1**, ..., **Pn** sont ses paramètres et **EXP** est une expression.

Il s'agit d'une **liaison globale** et donc aussi d'une **déclaration**.

Tout appel à la fonction **ID** possède comme valeur la valeur de **EXP** dans laquelle les occurrences libres de ses paramètres sont remplacées par les arguments.

Exemple

Voici une définition de fonction :

```
# let oppose x = -x;;  
val oppose : int -> int = <fun>
```

Voici quelques explications sur la réponse de l'interpréteur :

- `val oppose` informe qu'on a lié au nom `oppose` une valeur ;
- `: int -> int` informe que cette valeur est de type `int -> int` ;
- `= <fun>` est un affichage générique de la fonction.

Pour **appliquer une fonction à des arguments**, nous utilisons la construction syntaxique

`ID A1 ... An`

Les arguments sont séparés par des **espaces** sans utiliser de parenthèses englobantes. Elles servent uniquement à délimiter un argument qui serait une expression non atomique.

Exemples

Voici des applications de `max` sur des paramètres atomiques :

- `max 0 1 ;`
- `let x = 8 in max 3 x .`

Voici des applications de `max` sur des paramètres non atomiques :

- `max (1 + 1) 2 ;`
- `max (1 + max 1 3) (4 * 3) .`

Ainsi, réaliser un appel par `ID(A1, ..., An)` de manière similaire à ce qu'il se fait dans certains langages de programmation produira (souvent) une **erreur**.

La fonction `ID` serait dans ce cas appliquée au **n-uplet** `(A1, ..., An)`, ce qui n'est pas forcément recherché.

Exemple

Écrire `max (2, 3)` ou même `max(2, 3)` reviendrait à appliquer la fonction `max` sur l'unique argument couple `(2, 3)`.

Exemple

Considérons la fonction

```
# let sum_squares x1 x2 = x1 * x1 + x2 * x2;;  
val sum_squares : int -> int -> int = <fun>
```

Lors de l'appel `sum_squares 5 (2 + 1)`, les paramètres `x1` et `x2` sont substitués respectivement dans l'expression définissant la fonction par les arguments `5` et `2 + 1` (ou leurs valeurs `5` et `3`).

L'évaluation donne ainsi

$$\text{sum_squares } 5 \ (2 + 1) \rightarrow (5) * (5) + (2 + 1) * (2 + 1) \rightarrow 34$$

ou

$$\text{sum_squares } 5 \ (2 + 1) \rightarrow \text{sum_squares } 5 \ 3 \rightarrow (5) * (5) + (3) * (3) \rightarrow 34$$

suivant la **stratégie d'évaluation du langage** (ce point sera vu en détail plus loin).

Une **fonction locale** est une fonction définie à l'intérieur d'une autre. Sa **portée** s'étend à la fonction dans laquelle elle est définie. Elle suit les mêmes règles d'ombrage que celles qui s'appliquent aux liaisons locales de noms.

Exemple

```
# let f u =
  let aux u =
    u ^ "a" ^ u
  in
  (aux u) ^ (aux ("b" ^ u));;
val f : string -> string = <fun>
```

```
# f "";;
- : string = "abab"

# f "cd";;
- : string = "cdacdbcdabcd"
```

Il est également possible de réaliser des définitions de fonctions (locales) **simultanées**.

Exemple

```
# let f x =
  let g y = y - 2 = x
  and h x = 2 * x in
  g (h x);;
val f : int -> bool = <fun>
```

```
# f 1;;
- : bool = false

# f 2;;
- : bool = true
```

Observons ici que le paramètre `x` de `f` est visible et utilisable dans la fonction `g`.

Comme `h` a un paramètre `x`, le `x` de `f y` est ombré.

3.2.3. Types fonctionnels

Considérons une fonction de la forme

```
let f x = EXP
```

Lors de son évaluation, l'interpréteur affiche une réponse de la forme

```
val f : t1 -> t = <fun>
```

Le type associé au nom `f` est ainsi `t1 -> t`. Cela signifie qu'il s'agit d'une fonction acceptant un argument de type `t1` et renvoyant une valeur de type `t`.

Le symbole `->` est un **constructeur de types** : c'est en particulier le constructeur des **types fonctionnels**.

L'application de la fonction `f` à un argument `a` est **correctement typée** si

- l'argument `a` est de type `t1` ;
- l'expression `f a` est utilisée là où une valeur de type `t` est attendue.

Exemples

Considérons la fonction

```
# let f x =  
  if x >= 0 then  
    "positif"  
  else  
    "negatif";;  
val f : int -> string = <fun>
```

et les utilisations suivantes :

```
# "Nombre " ^ (f 21);;  
- : string = "Nombre positif"
```

Cette expression est **bien typée**.

À la fois ses arguments et son contexte sont de types attendus.

```
# 4 + (f 21);;  
Error: ... type string but ... expected of type int
```

Cette expression est **incorrectement typée**.

Son contexte n'est pas du type attendu.

Considérons une fonction de la forme

```
let f x1 x2 = EXP
```

Lors de son évaluation, l'interpréteur affiche une réponse de la forme

```
val f : t1 -> t2 -> t = <fun>
```

Ce type `t1 -> t2 -> t` doit se comprendre comme étant le type `t1 -> (t2 -> t)` car le constructeur `->` associe de la droite vers la gauche.

Ceci suggère que `f` est une fonction qui

- accepte un argument de type `t1` ;
- renvoie une valeur de type fonctionnel `t2 -> t`.

Une fonction définie avec deux paramètres est ainsi une **fonction à un paramètre** qui **renvoie une fonction** à un paramètre.

Exemple

Considérons la fonction

```
# let sum_squares x1 x2 =  
    x1 * x1 + x2 * x2;;  
val sum_squares : int -> int -> int = <fun>
```

Ceci suggère qu'il est possible d'appeler `sum_squares` avec un seul argument, au lieu de deux :

```
# sum_squares 0;;  
- : int -> int = <fun>
```

C'est effectivement bien le cas, et nous obtenons par `sum_squares 0` une fonction paramétrée par un entier et renvoyant un entier.

Nous allons explorer la sémantique de cela dans la prochaine partie.

3.2.4. Applications partielles et fonctions curryfiées

Cette conceptualisation des fonctions à deux paramètres se généralise à $n \in \mathbb{N}$ paramètres.

Considérons une fonction de la forme

```
let f x1 ... xn = EXP
```

Lors de son évaluation, l'interpréteur affiche une réponse de la forme

```
val f : t1 -> ... -> tn -> t = <fun>
```

Ce type est compris comme le type totalement parenthésé

```
t1 -> (t2 -> (t3 -> ... -> (tn -> t)...))
```

Comme cas particulier, quand $n = 0$, le type de `f` est simplement `t`.

Lors de l'application `f a1 ... ak` de `f` à $k \in \mathbb{N}$ arguments tels que $k \leq n$, nous obtenons une expression de type

`tk+1 -> ... -> tn -> t`

Lorsque $k < n$, il s'agit d'une **application partielle**.

Conséquence 1 : appliquer partiellement une fonction à des arguments **crée une nouvelle fonction**.

Conséquence 2 : l'application d'une fonction à un argument **associe de la gauche vers la droite**. Ainsi, l'application écrite de manière conventionnelle

`f a1 a2 ... ak`

est comprise comme l'application totalement parenthésée

`(...((f a1) a2)...) ak`

Exemple

Considérons la fonction ci-contre possède **exactement** trois paramètres :

```
# let surround pref suff fact =
    pref ^ fact ^ suff;;
val surround : string -> string -> string -> string =
    <fun>
```

Voici quelques applications possibles :

```
# surround "a" "b" "c";;
- : string = "acb"
```

```
# surround "(" " " " ";;
- : string -> string = <fun>
```

Il s'agit ici d'une application avec $k = 3$ selon les notations de la page précédente.

Il s'agit ici d'une application avec $k = 2$ et donc d'une application partielle.

Il y a plusieurs façons d'utiliser cette dernière fonction obtenue par application partielle :

1. directement par `(surround "(" " ") "abc"`, qui est équivalent à `surround "(" " ") "abc"` ;
2. via un alias : `let surround' fact = surround "(" " ") fact in surround' "abc"`.

Ainsi, `surround "(" " ")` est une fonction acceptant une chaîne de caractères et qui la renvoie avec `((` au début et `)` à la fin.

En OCaml (ainsi que dans beaucoup d'autres langages fonctionnels), les fonctions sont **curryfiées**. Ceci signifie qu'une fonction à plusieurs entrées est conceptualisée comme une **fonction à une seule entrée et une seule sortie** (pouvant elle-même être d'un type fonctionnel).

Exemples

- La fonction `(+)` est de type `int -> int -> int`, équivalent au type `int -> (int -> int)`.
L'appel `(+) 1` est donc de type `int -> int` et est une fonction qui accepte un entier et qui renvoie son successeur.
- La fonction `List.cons` est de type `'a -> 'a list -> 'a list`, équivalent au type `'a -> ('a list -> 'a list)`.
L'appel `List.cons 0` est donc de type `int list -> int list` et est une fonction qui accepte une liste d'entiers `lst` et qui renvoie la liste dont la tête est `0` et la queue est `lst`.
- La fonction `String.sub` est de type `string -> int -> int -> string`, équivalent aux types `string -> int -> (int -> string)` et `string -> (int -> (int -> string))`.
Nous pouvons donc la voir de manière équivalente comme une fonction à trois entrées et qui renvoie une chaîne de caractères, comme une fonction à deux entrées et qui renvoie une fonction, ou encore comme une fonction à une entrée et qui renvoie une fonction.
Par exemple, `String.sub "abcdefg" 2` est une expression de type `int -> string`.

C'est précisément le fait que les fonctions sont **curryfiées** qui donne accès au mécanisme d'**application partielle**.

Ce procédé est extrêmement puissant et utile pour avoir du **code concis et lisible** (ceci sera illustré plus tard lors de l'étude des fonctions d'ordre supérieur).

Une idée pour anticiper un peu : pour multiplier par deux les éléments d'une liste d'entiers `lst`, il suffit d'appliquer à chaque élément de `lst` la fonction `( * ) 2`. Celle-ci est de type `int -> int` et renvoie le double de son argument.

Supposons que nous disposions d'une fonction `f` de type `t1 -> t2 -> t`.

Nous savons alors que `f` possède **au moins deux entrées** (il se peut que `t` soit fonctionnel et ainsi que l'on puisse voir `f` comme une fonction à strictement plus de deux entrées).

Une question légitime se pose : comment appliquer `f` partiellement de sorte à accéder uniquement à son 2e paramètre (sans toucher à son 1er comme c'est le cas via une application partielle) ?

Réponse : utiliser la fonction `Fun.flip` de la bibliothèque standard. Son type est `('a -> 'b -> 'c) -> 'b -> 'a -> 'c` et sera expliqué plus tard. L'appel `Fun.flip f a2` est la fonction de type `t1 -> t` qui se comporte comme `f` dans le cas où son 2e argument est **spécialisé** à `a2`.

Exemple

```
# let f' = Fun.flip (-) 1;;  
val f' : int -> int = <fun>
```

```
# f' 2;;  
- : int = 1
```

3.2.5. Fonctions anonymes

Une fonction n'a pas nécessairement besoin d'avoir un nom pour exister.

Une fonction bien définie mais qui n'a pas de nom est une fonction anonyme.

La construction syntaxique

fun P1 ... Pn -> EXP

où P1, ..., Pn sont des paramètres et EXP est une expression permet de définir une fonction anonyme.

Exemples

```
# (fun a b -> (a + b) * a) 4 3;;  
- : int = 28
```

Définit une fonction anonyme appliquée aux arguments 4 et 3.

```
# let produit k =  
  fun x -> x * k;;  
val produit : int -> int -> int = <fun>
```

L'expression produit 10 a pour valeur une fonction de type
int -> int qui multiplie par 10 son argument.

La véritable manière de définir une fonction en OCaml se fait par

```
function P -> EXP
```

Cette expression est une fonction anonyme à un paramètre `P`.

Les deux autres manières de définir les fonctions sont du sucre syntaxique :

1. la construction

```
fun P1 P2 ... Pn -> EXP
```

est équivalente à

```
function P1 -> function P2 -> ... -> function Pn -> EXP
```

Ceci utilise le fait que les fonctions sont curryfiées.

2. La construction

```
let F P1 ... Pn = EXP
```

est équivalente à

```
let F = fun P1 ... Pn -> EXP
```

3.2.6. Inférence des types des fonctions

Le mécanisme d'inférence des types de OCaml agit statiquement et implicitement.

Il se charge de deviner le type d'une fonction en analysant son code.

Exemples

1.

```
let mystere x y z =  
  if x && (y 1) then z
```

Cette fonction est de type
`bool -> (int -> bool) -> unit -> unit .`

2.

```
let etrange x y =  
  "a" ^ ((y 1) ((x 'a') + 1)) ^ "b"
```

Cette fonction est de type
`(char -> int) -> (int -> int -> string) -> string .`

3.

```
let saugrenu x y z t =  
  1. +. (x ((y (not z) (t - 1)) ^ "ab") y)
```

Cette fonction est de type
`(string -> (bool -> int -> string) -> float) -> (bool -> .
int -> string) -> bool -> int -> float`

4.

```
let bizarre y =  
  ((fun x -> x + 1) 2) + (y (string_of_int 3))
```

Cette fonction est de type
`(string -> int) -> int .`

Il est possible de spécifier les types de certains des paramètres et/ou de la sortie pour **gagner en lisibilité** ou pour **restreindre les types** autorisés par

```
let F (P1 : T1) ... (Pn : Tn) : T = EXP
```

où les `T1`, ..., `Tn` et `T` sont des types.

Exemples

1. Sans restriction :

```
# let pair x y = (x, y);;  
val pair : 'a -> 'b -> 'a * 'b = <fun>
```

2. Avec restriction du type du 1er paramètre :

```
# let pair (x : string) y = (x, y);;  
val pair : string -> 'a -> string * 'a = <fun>
```

3. Avec restriction du type de retour :

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

3.2.7. Retardateurs

Considérons la liaison d'un nom à une valeur

```
let f = EXP
```

Rappelons que `f` peut être vue comme la fonction constante qui renvoie la valeur de l'expression `EXP`.

Cette déclaration fait que le calcul demandé pour l'évaluation d'`EXP` est réalisé **avant tout appel** potentiel à `f`.

Exemples

Ce programme imprime seulement `*\n` (l'opérateur `;` sert à enchaîner les expressions, il sera détaillé plus loin) :

```
let f =  
  print_string "*\n"  
let () =  
  f; f
```

Pire encore, ce programme imprime `*\n` :

```
let f =  
  print_string "*\n"  
let () =  
  ()
```

Comparons ceci avec la liaison

```
let f () = EXP
```

Ici, `f` peut être vue comme la fonction unaire constante qui renvoie la valeur d' `EXP` .

Cette déclaration ne demande pas l'évaluation de `EXP` . Cette expression n'est évaluée qu'au moment où `f` est utilisée via l'appel `f ()` .

Cette fonction de type `unit -> T` où `T` est le type d' `EXP` est un **retardateur** .

Cette construction est utile en particulier lorsque `EXP` effectue un **effet secondaire** et que celui-ci doit être exécuté à un moment précis.

Exemples

Ce programme imprime bien `*\n*\n` :

```
let f () =  
  print_string "*\n"  
let () =  
  f (); f ()
```

Ce programme n'imprime bien rien du tout :

```
let f () =  
  print_string "*\n"  
let () =  
  ()
```