INF6120 --- Programmation fonctionnelle et logique

Samuele Giraudo

Département d'informatique, Université du Québec à Montréal

giraudo.samuele@uqam.ca

Baccalauréat en informatique et génie logiciel

Automne 2025

1. Introduction

1.1. Informations générales

1.1.1. Informations pratiques

- Titre du cours : *Programmation fonctionnelle et logique*
- Sigle : INF6120
- Département : Informatique
- Enseignant : Samuele Giraudo (coordinateurs : Samuele Giraudo et Quentin Stiévenart)
- Courriel : giraudo.samuele@uqam.ca
- Bureau : PK-4430
- Page personnelle : https://igm.univ-mlv.fr/~giraudo/Home.html
- Site du cours : https://igm.univ-mlv.fr/~giraudo/Teaching/INF6120/2025-09/INF6120.html
- Plan de cours : https://info.uqam.ca/plan_cours/Automne%202025/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. **semaine 8** : examen 1, 50% ;
2. **semaine 15** : examen 2, 50%.

Contenu général :

- l'**examen 1** porte sur ce qui a été vu dans les 7 premières séances (programmation fonctionnelle) ;
- l'**examen 2** porte sur ce qui a été vu dans les 14 premières séances (programmation fonctionnelle et programmation logique).

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

Pour les examens, **4 feuilles recto-verso** au format A4 (ou équivalent) et manuscrites sont autorisées. Les noms et prénoms doivent être écrits sur **chaque page en haut à droite**.

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 programmes.
   - ☐ Typage statique.
2. **Programmation logique.**
   - ☐ Bases théoriques.
   - ☐ Unification et résolution.
   - ☐ Manipulation des listes.

1.1.3. Références

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

1. M. R. Clarkson, OCaml Programming : Correct + Efficient + Beautiful, 2021, https://cs3110.github.io/textbook/cover.html.
2. P. Flach, Simply Logical, 2022, https://book.simply-logical.space/src/simply-logical.html.

Elles sont consultables librement aux liens indiqués.

Quelques références utiles :

1. Y. Minsky, A. Madhavapeddy, Real World OCaml, 2nd Edition, 2022, https://dev.realworldocaml.org/.
2. X. Leroy *et al.*, The OCaml system release 5.3, 2025, https://ocaml.org/manual/5.3/index.html

Quelques autres références, 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

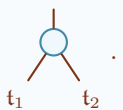
Un *arbre binaire* t est

1. soit une feuille

$$t = \; ;$$

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

$$t = \quad .$$



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.

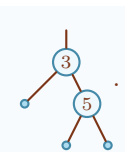
En OCaml, ceci peut se faire via la **définition du type**

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

L'expression

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

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



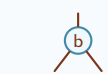
L'**algorithme d'insertion dans un arbre binaire de recherche** admet la description suivante.

Soit t arbre binaire de recherche et a une valeur. L'arbre $t \leftarrow a$, obtenu en **insérant** a dans t est défini par

1. si $t =$  , alors $t \leftarrow a :=$ 

;
2. sinon, t est de la forme $t =$ 

où b est une valeur. Dans ce cas,

   si $a \leqslant b$, alors $t \leftarrow a :=$ 

   et 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 `btree`. Il s'agit d'un arbre binaire de recherche ayant 5 nœuds internes.

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** :

- 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** :

$$\texttt{one\_of\_two [0; 1; 2; 3; 4]} \longrightarrow \texttt{0 :: one\_of\_two [2; 3; 4]} \longrightarrow \texttt{0 :: 2 :: one\_of\_two [4]} \longrightarrow \texttt{0 :: 2 :: [4]} \rightsquigarrow \texttt{[0; 2; 4]}$$

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. La définition de **fonctions** permet d'expliquer comment réaliser le calcul.

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.

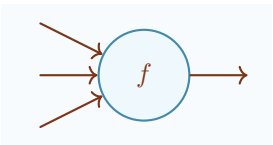
  ⇝ en pratique :

« 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, 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, on n'utilise pas de boucles `while`, `do while`, `for` ou autres).

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

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 de la 1re occurrence de `x` dans `lst`.

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

```python
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

```ocaml
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
```

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, Pos1),
    Pos is Pos1 + 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`.

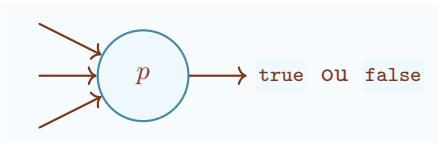
Ceci est un petit défaut dans notre implantation.

Dans ce paradigme, il est possible de laisser le 3e 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.`

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 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.

1.3. Initiation rapide à l'OCaml

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 secondaires** en programmation fonctionnelle. Seule la valeur compte.
7. Cependant, nous allons assouplir la dernière règle 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 = 16 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**.

- *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`.

Exemple

```
let sum_squares x y =
    x * x + y * y

let () =
    let x = sum_squares 8 (sum_squares 2 4) 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.

- *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 $f : \{1,2,3,4\} \to \{1,2\}$ telle que $f(1) = 1$, $f(2) = 2$, $f(3) = 2$ et $f(4) = 2$.

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`.
- `[]` 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 appelle 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.

Exemple

```
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`.

Exemple

```
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.

Exemple

```
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

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 : 1ers 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).

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 ».

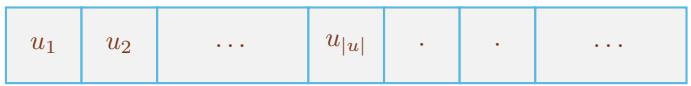
Une machine de Turing est un quadruplet (E, i, t, Δ) où

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. $\Delta : E \times \{\cdot, 0, 1\} \to E \times \{\cdot, 0, 1\} \times \{\mathrm{G}, \mathrm{D}\}$ est une fonction de transition.

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

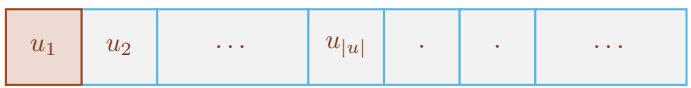
L'*exécution* de M sur u consiste à :

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



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

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



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

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

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

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 $s' = \mathrm{D}$, déplacer la tête de lecture/écriture d'un pas vers la droite ; sinon, la déplacer si possible d'un pas vers la gauche.
5. Si $e = t$, alors l'exécution est terminée ; sinon, revenir en 1.

Le *résultat* de l'exécution de M sur u est le mot $M(u)$ défini comme étant **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 $\cdot$).

Exemple

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

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

Registre d'état	Ruban
e_1	0 0 1 · · ⋯
e_1	1 0 1 · · ⋯
e_1	1 1 1 · · ⋯
e_1	1 1 0 · · ⋯
e_2	1 1 0 · · ⋯

Voici les étapes du calcul de $M(001)$.

L'exécution de M sur u fournit ainsi le résultat $M(001) = 110$.

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

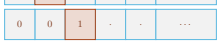
Il est possible de construire des machines de Turing telles que pour certaines entrées u, l'exécution de M sur u **ne se termine pas**.

Exemple

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

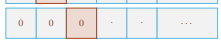
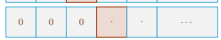
$$(e_1, 0) \mapsto (e_1, 0, \mathrm{D}), \qquad (e_1, 1) \mapsto (e_2, 1, \mathrm{G}), \qquad (e_1, \cdot) \mapsto (e_1, \cdot, \mathrm{D}).$$

Calcul de $M(001)$:

Registre d'état	Ruban
e_1	
e_1	
e_1	
e_2	

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

Calcul de $M(000)$:

Registre d'état	Ruban
e_1	
e_1	
e_1	
e_1	
e_1	

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

Un langage de programmation 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 L une **fonction d'émulation** `emul`, paramétrée par le codage d'une machine de Turing M et le codage d'un mot $u \in \{0,1\}^*$, et qui renvoie le codage de $M(u)$.

Ceci est un exercice très facile pour des **langages impératifs** comme C, PYTHON ou JAVA. Ils sont donc Turing-complets. (Note : 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.
    ...
    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).

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 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 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

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

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

$$v = 01100101\ 01110100 \cdots 00101001$$

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

En décimal, ce nombre est

$$\mathrm{code}(M) =$$

$$11952734835224639476981884285665739948629390562908017629035981357571402948738810055002606104372079574033 72$$

$$26979189145773772473673716516841910132913242275141863782405111884164058543999074736181406429955364904425 2$$

$$29975166545029466892832412634123268241667345353999388604458150336894449185720558624721864880882898175696 9.$$

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 $M \leqslant M'$ si $\text{code}(M) \leqslant \text{code}(M')$.

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

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

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, $P(\epsilon) = \text{oui}$, $P(1) = \text{non}$;
- 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 (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

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

qui est intuitivement calculable.

Exemples

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

Ces fonctions sont récursives au sens précédent.

La dernière est même récursive au sens usuel.

Exemple

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

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

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

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. $\lambda x.\, \underline{x}$
2. $\lambda x.\, y$
3. $\lambda x.\, \lambda y.\, (\underline{y}\, \underline{x})$
4. $x\, (\lambda x.\, y\, (\lambda y.\, (\underline{x}\, \underline{y})))$
5. $(\lambda x.\, y\, \underline{x})\, (\lambda y.\, \underline{y}\, x)$
6. $\lambda z.\, (\lambda x.\, (y\, \underline{x})\, \underline{z})\, (\lambda y.\, (\underline{y}\, x)\, \underline{z})$

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 $(\lambda x.f)\,a \;\to\; f[x := a]$.

Exemple

$$(\lambda x.\,(x\,y)\,x)\,(z\,z) \;\to\; ((z\,z)\,y)\,(z\,z)$$

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 $e_1 \to e_2$.

Exemple

$$x\,((\lambda x.\,(x\,y)\,x)\,(z\,z)) \;\to\; x\,(((z\,z)\,y)\,(z\,z))$$

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 :

$$((\lambda x.\, \lambda y.\, x)\, u)\, v \;\to\; (\lambda y.\, u)\, v \;\to\; u$$

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

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 `add` acceptant deux entiers et renvoyant un entier, l'expression `add(10, "3")` 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,

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

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

Cette information est donnée lors de la compilation :

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

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

Considérons le programme OCaml, similaire au précédent,

```
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)
```

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 résolus 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 quelques 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

3. Programmation fonctionnelle

3.1. Expressions de base

3.1.1. Généralités

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

Nous aurons besoin (au minimum) des logiciels 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/5.1/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**.

L'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.

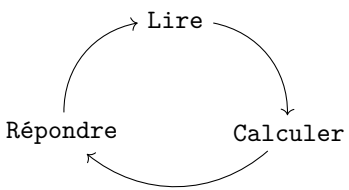
3.1.2. Interpréteur

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

```
Objective Caml version 5.3.0
...
#
```

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'entree ;
- 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`.

Exemples

```
# 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.

```
# 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`.

Exemples

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

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

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

Le nom `s` est lié à la valeur `9`.

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

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

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

Exemples

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

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

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

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

L'évaluation de cette phrase produit une erreur lors de son interprétation.

En effet, le nom `x` de la l. 2 n'est pas défini.

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.

Exemples

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

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

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

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

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

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

Voici quelques exemples qui illustrent quelques particularités des liaisons simultanées.

Exemples

```
# 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.

```
# 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.

Exemples

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

Il y a trois liaisons simultanées.

L'occurrence du nom `x` en l. 4 fait référence à la définition de `x` en l. 3. Cette définition n'influe pas sur la définition de `x` en l. 1.

```
# let x = 1
  and y = x + 1;;
error: unbound value x
```

L'évaluation de cette phrase produit une erreur lors de son interprétation.

En effet, le nom `x` de la l. 2 n'est pas défini.

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**.

Exemple

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

- 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 in x + 1;;
- : int = 4
```

3.1.4. Types de base

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

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

Voici les *opérations booléennes* :

Opérateur	Arité	Rôle
`not`	1	Non logique
`&&`	2	Et logique
`||`	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

- `0`, `1024`, `-82`
- `0x0` (entier 0), `0x400` (entier 1024), `-0xAE00F23` (entier -182456099)
- `0b1011011` (entier 91), `-0b101` (entier -5)

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

Sur un système `64` bits, ceci va de

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

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
`-`, `+`	1	Moins, Plus (signe)
`-`, `+`	2	Soustraction, Addition
`/`, `*`	2	Division, Multiplication
`succ`	1	Successeur
`pred`	1	Prédécesseur
`abs`	1	Valeur absolue
`mod`	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
`=`, `<>`	2	Égalité, Différence
`<`, `>`	2	Comparaison stricte
`<=`, `>=`	2	Comparaison large

Ces opérateurs produisent des valeurs de type `bool`.

Exemples

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

```
# true = (false || false);;
- : 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`.

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

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

Ces opérateurs produisent des valeurs de type `int`.

Exemples

```
# 1 lsl 10;;
- : int = 1024
```

```
# (lnot 0) lsr 1;;
- : int = 4611686018427387903
```

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

```
# 0;;
- : int = 0
```

```
# 0.;;
- : float = 0.
```

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

Sur un système `64` bits, ceci va de

$$-\texttt{max\_float} = -1.79769313486231571 \times 10^{308} \quad \text{à} \quad \texttt{max\_float} = 1.79769313486231571 \times 10^{308}.$$

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
`abs_float`	1	Valeur absolue
`floor`, `ceil`	1	Partie entière inf., sup.
`sqrt`	1	Racine carrée
`log`, `exp`	1	Logarithme népérien, exponentielle
`cos`, `sin`, `tan`	1	Fonctions trigonométriques
`**`	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.

Exemples

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

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

Exemple

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

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

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 de les utiliser au 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.

Rappellons 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` → `if true then 21 else 24` → `21`,
- `if if "ab" = "ab" then 1 = 0 else true then 28 else 21`
  → `if if true then 1 = 0 else true then 28 else 21`
  → `if 1 = 0 then 28 else 21` → `if false then 28 else 21` → `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

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`.

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 $f : E \to E'$.

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 $f(x)$ l'élément x'.

Une fonction à n **entrées** est une fonction $f : (E_1 \times \cdots \times E_n) \to E'$. Nous notons alors par $f(x_1, \ldots, x_n)$ l'élément $f((x_1, \ldots, x_n))$.

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

Nous verrons cependant que les fonctions à plusieurs entrées sont conceptualisées de manière beaucoup plus ingénieuse en programmation fonctionnelle.

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 $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ 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, ..., a_n, nous disons que l'on *applique* f *à* a_1, ..., 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 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.

Réaliser un appel par `ID(A1, ..., An)` de manière similaire à ce qu'il se fait dans certaines langages de programmation produira (souvent) une **erreur**. La fonction `ID` serait dans ce cas appliquée à un **unique** `n`**-uplet** `(A1, ..., An)`, ce qui n'est pas forcément recherché.

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 respecivement dans l'expression définissant la fonction par les arguments, `5` et `2 + 1` (ou leurs valeurs `5` et `3`).
L'évaluation donne ainsi

`sum_squares 5 (2 + 1)` → `(5) * (5) + (2 + 1) * (2 + 1)` → `34`

ou

`sum_squares 5 (2 + 1)` → `sum_squares 5 3` → `(5) * (5) + (3) * (3)` → `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'intérieure 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
```

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>
```

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

Cette expression est bien typée.

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

Cette expression est incorrectement typée.

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>
```

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 tel que $k \leqslant n$, 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, `f a1 a2 ...ak` est compris comme l'application totalement parenthésée `(...((f a1) a2) ...) ak`.

Exemple

Considérons la fonction

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

Cette fonction possède exactement trois paramètres. Voici quelques applications possibles :

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

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

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

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;;
val surround' : string -> string = <fun>
# surround' "abc";;
- : string = "((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 équivalence 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 le 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).

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 type** de l'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

Sans restriction :

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

Avec restriction du type du premier 1er paramètre :

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

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`.

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 d'`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 lorsqu'`EXP` effectue un **effet secondaire** et que celui-ci doit être exécuté à un moment précis.

3.3. Entrées, sorties et séquences

3.3.1. Entrées/sorties

Les interpréteurs et leurs boucles d'interaction suffisent pour définir des fonctions et les tester. Les résultats sont affichés par l'interpréteur lui-même.

Cependant, il est important d'avoir à disposition de véritables fonctions d'entrée/sortie lorsqu'il est question de développer des programmes complets.

Il existe pour cela des **fonctions d'entrée/sortie** qui permettent de lire des données sur l'entrée standard et d'en écrire sur la sortie standard.

Les fonctions d'entrée/sortie utilisent le type `unit` et son unique valeur `()`.

Rappel : l'utilisation d'entrées/sorties fait **sortir du paradigme de programmation fonctionnelle pure**. Elles produisent des effets secondaires (impression sur la sortie standard ou bien attente d'une action de l'utilisateur).

En principe, les fonctions d'écriture ne renvoient rien. C'est leur **effet secondaire** qui forme leur intérêt.

En pratique, comme toute fonction doit renvoyer une valeur, par convention, **toute fonction d'écriture renvoie** `()`.

Les fonctions d'écriture principales sont

- `val print_int : int -> unit`
- `val print_float : float -> unit`
- `val print_char : char -> unit`
- `val print_string : string -> unit`
- `val print_newline : unit -> unit`
- `val print_endline : string -> unit`

La fonction d'**impression formatée**

```
val Printf.printf : ('a, out_channel, unit) format -> 'a
```

fonctionne comme le `printf` du langage C.

Exemple

La phrase

```
# Printf.printf "J'ai %d pommes et %d %s" 3 (3 + 1) "bananes";;
```

imprime `J'ai 3 pommes et 4 bananes`.

En principe, les fonctions de lecture ne prennent rien en entrée. C'est ce qu'elles renvoient, c'est-à-dire la **valeur saisie** par l'utilisateur, qui forme leur intérêt.

En pratique, comme toute fonction qui ne possède pas de paramètre est une constante, il n'est pas possible de les implanter ainsi. Pour cette raison, ces fonctions sont bâties comme des **retardateurs**.

Les fonctions de lecture principales sont

- `val read_int : unit -> int`
- `val read_float : unit -> float`
- `val read_line : unit -> string`

La fonction de **lecture formatée**

```
val Scanf.scanf : ('a, 'b, 'c, 'd) Scanf.scanner
```

fonctionne comme le `scanf` du C, à la différence près qu'elle ne réalise pas d'affectation des valeurs lues à des variables mais effectue une action selon une fonction.

Exemple

La phrase

```
# Scanf.scanf "%d %d %d" (fun n1 n2 n3 -> print_int (n1 + n2 + n3));;
```

lit trois entiers sur l'entrée standard et renvoie la valeur renvoyée par la fonction anonyme mentionnée appliquée aux valeurs lues. Cette fonction affiche leur somme et renvoie `()`.

3.3.2. Séquences

L'utilisation des fonctions d'entrée/sortie est parfaitement adaptée à la programmation impérative dans laquelle les exécutions des instructions s'enchaînent les unes après les autres.

Dans notre contexte, pour les utiliser, nous avons besoin de pouvoir **enchaîner** deux expressions l'une à la suite de l'autre. On utilise pour cela l'*opérateur de séquence* `;`. Il s'utilise de la manière suivante :

```
EXP1; EXP2
```

où `EXP1` est une expression dont la valeur est de type `unit` et `EXP2` est une expression. La **valeur** d'`EXP1; EXP2` est celle d'`EXP2`.

L'opérateur `;` **associe de la gauche vers la droite**. Ainsi,

```
E1; E2; ... ; En
```

est compris comme l'expression totalement parenthésée

```
(...(E1; E2); ...); En
```

La **valeur** de `E1; E2; ... ; En` est ainsi celle de `En`. De ce fait, les expressions `E1`, ..., `En-1` doivent être de type `unit`.

Exemples

```
# let add x y =
      print_string "Appel de add";
      print_int x;
      print_int y;
      x + y;;
val add : int -> int -> int = <fun>
```

Tout appel `add a b` imprime des éléments sur la sortie standard puis renvoie `a + b`.

```
# let test_div n =
      if n mod 2 = 0 then
          print_string "pair\n";
      if n mod 3 = 0 then
          print_string "multiple de 3\n";;
val test_div : int -> unit = <fun>
```

Tout appel `test_div a` imprime une chaîne de caractères si `a` est divisible par 2 et une autre chaîne si `a` est divisible par 3. La valeur `()` est renvoyée dans tous les cas.

Le schéma

```
if COND then
    print_X;
```

où `COND` est une expression de type `bool` et `print_X` est une fonction d'impression est bien adapté à ce contexte. Il permet d'imprimer une information si une condition est vérifiée.

L'utilisation de l'opérateur de séquence `;` est source d'erreurs classiques.

Exemple

```
let div_decr x =
    if x mod 2 = 0 then
        print_string "pair";
        x / 2
    else
        print_string "impair";
        x - 1;;
```

Cette fonction se comprend comme la fonction ci-contre (en améliorant l'indentation) faisant apparaître clairement une **erreur de syntaxe** :

```
let div_decr x =
    if x mod 2 = 0 then
        print_string "pair";
    x / 2
    else
        print_string "impair";
    x - 1;;
```

L'opérateur de séquence est **moins prioritaire** que la conditionnelle.

Pour résoudre ce problème, nous utilisons la notion de bloc. Un *bloc* est une expression de la forme

`begin EXP end`

où `EXP` est une expression. La valeur de `begin EXP end` est celle d'`EXP`.

Exemple

```
let div_decr x =
    if x mod 2 = 0 then begin
        print_string "pair";
        x / 2
    end
    else begin
        print_string "impair";
        x - 1
    end
```

3.4. Écriture de projets

3.4.1. Méthode directe

Au lieu d'utiliser l'OCaml uniquement en mode interprété via `ocaml` ou `utop`, il est possible d'écrire de manière classique un programme dans un (ou plusieurs) fichier(s) au sein d'un projet et de le compiler pour **obtenir un exécutable**.

Un *projet* est un ensemble de fichiers `.ml` dont exactement un contient une fonction principale et d'éventuels autres fichiers de configuration.

Un projet tenant sur un seul fichier `Program.ml` se compile par la commande

```
ocamlc -o Program Program.ml
```

Ceci invoque le **compilateur bytecode**. Le code produit a l'avantage d'être portable.

Le **compilateur natif** peut être invoqué via la commande

```
ocamlopt -o Prog Prog.ml
```

Le code produit a le désavantage de dépendre de l'architecture de la machine qui l'a compilé mais a l'avantage d'être performant.

Pour **compiler un projet**,

1. on construit un *fichier objet* (`.cmo` ou `.cmx`) pour chaque fichier `F.ml` du projet au moyen de la commande

   ```
   ocamlc -c F.ml
   ```

   ou bien

   ```
   ocamlopt -c F.ml
   ```

2. on appelle l'*éditeur de liens* par la commande

   ```
   ocamlc -o Prog F1.cmo ... Fn.cmo
   ```

   ou bien

   ```
   ocamlopt -o Prog F1.cmx ... Fn.cmx
   ```

   où les `F1.cm*`, ..., `Fn.cm*` sont les fichiers objets du projet.

Dans le cas où un fichier `B.ml` a besoin d'une entité `x` définie dans un fichier `A.ml` du projet, il faut utiliser `x` aux endroits désirés dans `B.ml` en la **préfixant** par `A.`, formant `A.x`.

Exemple

```
(* A.ml *)

let double x = 2 * x
```

```
(* B.ml *)

let quad x = 2 * (A.double x)
```

Dans ce projet, la fonction `double` de `A.ml` s'emploie dans `B.ml` via l'identificateur `A.double`.

Il est possible de réaliser une **ouverture explicite** de `A.ml` dans `B.ml` par

```
open A
```

permettant une utilisation directe (sans préfixe) des entités de `A.ml` dans `B.ml`.

Exemple

```
(* A.ml *)

let double x = 2 * x
```

```
(* B.ml *)

open A

let quad x = 2 * (double x)
```

L'ouverture explicite de `A.ml` dans `B.ml` permet d'appeler la fonction `double` dans `B.ml` sans le préfixe `A.`.

Un *espace de nom* est une fonction (au sens mathématique) qui a un ensemble de symboles (des identificateurs) associe leur définition. Les espaces de noms permettent de **restreindre la visibilité** de certaines définitions.

À chaque fichier `.ml` est associé un espace de noms qui lui est propre.

Il est ainsi possible d'avoir dans un même projet deux fichiers qui définissent des fonctions nommées par un même identificateur.

Exemple

```
(* A.ml *)

let fct x =
    ...
```

```
(* B.ml *)

let fct x y =
    ...
```

```
(* C.ml *)
...
A.fct 1
...
B.fct 'a' 2
...
```

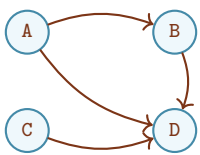
Dans ce projet, deux fonctions nommées `fct` sont définies.
Leur nom absolu n'est en revanche pas le même (`A.fct` et `B.fct`).
Il n'y a ainsi aucune ambiguïté dans `C.ml` qui utilise ces deux fonctions.

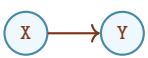
À la différence de certains autres langages, il est impossible de construire le fichier objet d'un fichier `X.ml` qui utilise des entités de `Y.ml` avant d'avoir produit celui de `Y.ml`.

Il faut donc compiler le projet dans l'ordre dicté par un **tri topologique** du graphe dual du graphe d'inclusions du projet.

Exemple

Considérons le **graphe d'utilisations** suivant :



Toute flèche 

signifie que le fichier `X.ml` utilise des entités de `Y.ml`.

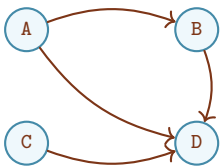
On a les trois ordres suivants possibles :

- `D.ml`, `C.ml`, `B.ml`, `A.ml` ;
- `D.ml`, `B.ml`, `C.ml`, `A.ml` ;
- `D.ml`, `B.ml`, `A.ml`, `C.ml`.

L'utilitaire `ocamldep` permet de calculer les dépendances des fichiers d'un projet en un format utilisable par `make`.

Exemple

Reprenons le graphe d'inclusions



La commande `ocamldep *.ml` affiche

```
A.cmo: D.cmo B.cmo
A.cmx: D.cmx B.cmx
B.cmo: D.cmo
B.cmx: D.cmx
C.cmo: D.cmo
C.cmx: D.cmx
D.cmo:
D.cmx:
```

Il est ainsi possible d'écrire des `Makefile` génériques en appelant à l'intérieur `ocamldep` de façon adéquate.

3.4.2. Utilisation de dune

Une autre façon, plus sophistiquée, de gérer des projets est offerte par l'outil `dune`.

Il permet, entre autres,

- de gérer la compilation d'un projet ;
- de gérer la mise en place de tests ;
- de gérer les dépendances du projet à des bibliothèques externes ;
- de concevoir des bibliothèques.

Pour **créer un nouveau projet** `NAME`, on utilise la commande

```
dune init project NAME
```

Ceci créé un répertoire `NAME` dans le répertoire courant.

Il contient divers fichiers et sous-répertoires. En particulier, il propose un sous-répertoire `bin` contenant un programme `main.ml` doté de la fonction principale.

Il y a aussi divers fichiers `dune`, utilisés pour la configuration de la construction du projet.

En se rendant dans le répertoire du projet (`cd NAME`), on le **construit** via la commande

```
dune build
```

Cela invoque le compilateur OCaml de manière adéquate pour produire un exécutable. Celui-ci a pour chemin relatif `_build/default/bin/main.exe`.

La commande

```
dune exec NAME
```

permet de l'**exécuter**.

Il y a aussi à la racine un fichier `dune-project` contenant des informations sur le projet (auteurs, description, version, *etc.*).

Le répertoire `lib` est propice pour y mettre les différents fichiers du projet, autre que celui qui contient la fonction principale.

Soit `PROJECT` un projet OCaml structuré via `dune`. En étant dans le répertoire `PROJECT` :

- pour **nettoyer le projet** en supprimant tous les fichiers qui peuvent être régénérés par `dune` :

```
dune clean
```

- pour **lancer un interpréteur** (et pouvoir utiliser les définitions du projet et les tester) :

```
dune utop
```

  Les différentes définitions sont accessibles dans `utop` via le préfixe `PROJECT.MODULE.` où `MODULE` est le nom du module dans lequel figure la définition souhaitée ;

- pour **installer localement** le projet (et pouvoir l'utiliser dans d'autres projets) :

```
opam pin add PROJECT .
```

  Il faut alors, dans les fichiers `dune` du projet utilisateur, mentionner la dépendance par l'expression `(libraries ... PROJECT)` ;

- pour **mettre à jour l'installation** du projet après une modification de celui-ci,

```
opam reinstall PROJECT
```

- pour imprimer la **liste des projets** installés :

```
opam list
```

3.5. Types

3.5.1. L'algèbre des types

Rappels :

- en programmation (fonctionnelle), un **type** désigne un **ensemble de valeurs** ;
- dire qu'une expression `EXP` est de type `T` est équivalent à dire qu'`EXP` **est un élément** de l'ensemble `T`.

Il existe deux sortes de types :

1. les *types scalaires*, qui sont des types atomiques et définis à l'avance dans le langage ;
2. les *types construits*, qui sont des constructions faisant intervenir des types scalaires ou construits et des **opérateurs de types**.

Dans certains langages de programmation, les types peuvent avoir des relations entre eux (par exemple, un type `T` peut être un sous-ensemble d'un type `S`).

L'ensemble des types d'un langage et des opérateurs de types est son *algèbre des types*.

La **définition d'un nouveau type** `ID` se fait par

```
type ID = OP
```

où `OP` fait intervenir des types et des opérateurs de types.

Rappelons que les **types scalaires** dont nous disposons sont

`int`, `float`, `char`, `string`, `bool`, `unit`.

Voici les *opérateurs de types* principaux que nous allons considérer (l'opérateur `->` a déjà été vu) :

Opérateur	Arité	Nom
`->`	2	Flèche
`*`	2	Produit cartésien binaire
`*`	$\geqslant 2$	Produit cartésien multiple
`{ }`	$\geqslant 1$	Produit nommé
`|`	$\geqslant 1$	Somme

3.5.2. Types produit

Étant donnés deux types `T1` et `T2`,

`T1 * T2`

désigne le type *produit cartésien binaire* de `T1` et `T2`.

Il contient pour valeurs les *couples* `(e1, e2)` où `e1` (resp. `e2`) est de type `T1` (resp. `T2`).

Exemple

Par la définition `type point = int * int` le type `point` contient tous les couples à coordonnées entières.

Un couple s'écrit par `(e1, e2)`. Les parenthèses sont facultatives.

Exemples

```
# (3.5, 21);;
- : float * int = (3.5, 21)
```

```
# 3.5, 21;;
- : float * int = (3.5, 21)
```

```
# (1, (2, 3));;
- : int * (int * int) = (1, (2, 3))
```

```
# ((1, 2), 3);;
- : (int * int) * int = ((1, 2), 3))
```

Si `c` est un couple, nous accèdons à sa 1re coordonnée par `fst c` et à sa 2e coordonnée par `snd c`.

Exemples

```
# let add c =
      (fst c) + (snd c);;
val add : int * int -> int = <fun>
```

```
# add (2, 4);;
- : int = 6
```

L'opérateur de types `*` est prioritaire face à `->`.

Il existe un moyen plus élégant (et plus général) pour accéder aux coordonnées de `c` au moyen de

```
let (c1, c2) = c in ...
```

Exemples

```
# let add c =
      let (c1, c2) = c in
      c1 + c2;;
val add : int * int -> int = <fun>
```

```
# let first' c =
      let (c1, _) = c in
      c1;;
val first' : 'a * 'b -> 'a = <fun>
```

L'opérateur de types `*` est **non associatif** : si `T1`, `T2` et `T3` sont trois types,

`(T1 * T2) * T3`

est un type différent de

`T1 * (T2 * T3)`

Exemple

```
# let conc_21 x =
      let (x1, x2) = x in
      let (x11, x12) = x1 in
      x11 ^ x2 ^ x12;;
val conc_21 : (string * string) * string -> string
    = <fun>
```

```
# conc_21 (("a", "b"), "c");;
- : string = "acb"
```

```
# let conc_12 x =
      let (x1, x2) = x in
      let (x21, x22) = x2 in
      x21 ^ x1 ^ x22;;
val conc_12 : string * (string * string) -> string
    = <fun>
```

```
# conc_12 ("a", ("b", "c"));;
- : string = "bac"
```

Les types `(string * string) * string` et `string * (string * string)` sont bien différents.

Pour tout $n \geqslant 3$, étant donnés des types `T1`, ..., `Tn`,

```
T1 * ... * Tn
```

désigne le type *produit cartésien multiple* de `T1`, ..., `Tn`.

Il contient pour valeurs les *n-uplets* `(e1, ..., en)` où pour tout $1 \leqslant i \leqslant n$, `ei` est de type `Ti`.

Exemple

```
type tr = int * unit * (int -> char)
```

Le type `tr` contient les triplets `(x1, x2, x3)` où `x1` est de type `int`, `x2` de type `unit` et `x3` est de type `int -> char`.

Un n-uplet s'écrit

```
(e1, ..., en)
```

Les parenthèses sont facultatives.

Exemple

```
# (0., 1, "abc", 'v');;
- : float * int * string * char = (0., 1, "abc", 'v')
```

Ceci construit un quadruplet.
Notons qu'il n'a pas été nécessaire de définir ce type au préalable.

Il est impossible d'utiliser `fst` et `snd` pour accéder aux coordonnées d'un n-uplet lorsque $n \neq 2$.

Si `p` est un tel n-uplet, nous accédons à ses coordonnées au moyen de

```
let (e1, ..., en) = p in ...
```

Exemple

```
# let p = (2.5, 3.4, -1.2) in
      let (x, y, z) = p in
          x +. z;;
- : float = 1.3
```

Il est possible de ne recueillir que la k^e de ses coordonnées par

```
let (_, ..., _, ek, _, ..., _) = p in ...
```

Exemple

```
# let (_, _, x, _, _) = (1, 2, 3, 4, 5) in x + 10;;
- : int = 13
```

Le symbole `_` est le *joker*. Il ne peut pas être lié à une valeur mais accepte néanmoins syntaxiquement d'être défini.

Pour tout $n \geqslant 1$, étant donnés des types `T1`, ..., `Tn` et des identificateurs `ID1`, ..., `IDn`,

```
{ID1 : T1 ; ... ; IDn : Tn}
```

désigne le type *produit nommé* de `T1`, ..., `Tn`.

Il contient pour valeurs les *enregistrements* dont les *champs* sont `ID1`, ..., `IDn` de types respectifs `T1`, ..., `Tn`.

Exemple

```
type personne = {nom : string ; age : int}
```

Un enregistrement s'écrit

```
{ID1 = EXP1 ; ... ; IDn = EXPn}
```

où `EXP1`, ..., `EXPn` sont des expressions de types respectifs `T1`, ..., `Tn`.

Exemple

```
# {nom = "Haskell Curry" ; age = 81};;
- : personne = {nom = "Haskell Curry"; age = 81}
```

Il est possible d'accèder au champ `c` d'un enregistrement `e` par

`e.c`

Exemple

En reprenant l'exemple précédent,

```
# let p = {nom = "Alan Turing" ; age = 41} in p.nom;;
- : string = "Alan Turing"
```

est une phrase qui s'évalue en le champ `nom` de la valeur `p` de type `personne` localement liée.

Voici une fonction qui accepte deux valeurs de type `personne` et qui renvoie le nom de la personne la plus âgée (et la chaîne de caractères vide en cas d'égalité) :

```
# let nom_du_plus_age p1 p2 =
      if p1.age > p2.age then
          p1.nom
      else if p1.age < p2.age then
          p2.nom
      else
          "";;
val nom_du_plus_age : personne -> personne -> string = <fun>
```

En programmation fonctionnelle, étant donné un nom `x`, il est **impossible de modifier la valeur** à laquelle `x` est lié. Il est seulement possible, via l'ombrage de nom, de lier un **nouveau** nom `x` à une nouvelle valeur.

Pour « modifier » un enregistrement, il faut en **reconstruire** un en prenant compte de la modification.

Si `e` est un enregistrement possédant (entre autres) des champs `c1`, ..., `cn`, l'expression

```
{e with c1 = v1 ; ... ; cn = vn}
```

est un enregistrement dont les valeurs des champs sont les mêmes que celles de ceux de `e`, sauf pour les champs `c1`, ..., `cn` dont les valeurs sont respectivement égales à `v1`, ..., `vn`.

Exemples

```
type point = {a : int ; b : int ; c : int}
```

```
# let p1 = {a = 1 ; b = 2 ; c = 3};;
val p1 : point = {a = 1; b = 2; c = 3}
```

```
# let p2 = {p1 with a = -1};;
val p2 : point = {a = -1; b = 2; c = 3}
```

```
let p3 = {p1 with b = 3 ; c = 4};;
val p3 : point = {a = 1; b = 3; c = 4}
```

Il est incorrect de définir des types enregistrements qui ont un même identificateur de champ.

Exemple

```
# type t1 = {a : int ; b : int}
type t1 = {a : int ; b : int; }
# type t2 = {a : int ; c : char}
type t2 = { a : int; c : char; }

# let f x = x.a
val f : t2 -> int = <fun>
```

Problème : le type de la fonction `f` ne peut pas être inféré correctement.

En effet, le champ `a` étant commun aux types `t1` et `t2`, le type de l'expression `x.a` ne peut pas être déterminé.

L'interpréteur accepte tout de même ces phrases mais il néglige la définition du type `t1` (qui est plus ancienne).

Ceci est correct :

```
# {a = 2 ; c = 'y'};;
- : t2 = {a = 2; c = 'y'}
```

Ceci provoque une erreur :

```
# {a = 2 ; b = 3};;
Error: The record field label b belongs to the type t1
       but is mixed here with labels of type t2
```

Il est ainsi non recommandé de définir, dans un même espace de noms, des types produit nommés ayant des noms de champs communs.

Cette contrainte **ne s'applique plus** si les types sont définis dans des **fichiers différents** car ils appartiennent à des **espaces de noms différents**.

Nous accèdons alors aux noms des champs d'un enregistrement en les préfixant par `A.` s'ils sont d'un type défini dans un fichier nommé `A.ml`.

Exemple

```
(* A.ml *)

type a = {a : int ; b : int}
```

```
(* B.ml *)

type b = {a : int ; c : char}
```

```
(* C.ml *)

let c1 = {B.a = 2 ; B.c = 'y'}
and c2 = {A.a = 2 ; A.b = 3}
```

3.5.3. Types somme

Étant donnés des identificateurs `Id1`, ..., `Idn` dont les 1res lettres sont des majuscules,

```
Id1 | ... | Idn
```

désigne le type *somme* de `Id1`, ..., `Idn`.

Il contient exactement `n` valeurs : `Id1`, ..., `Idn`.

Ces valeurs sont appelées *constructeurs*.

Exemple

```
type numero = Un | Deux | Trois
```

Une *valeur* d'un type somme s'écrit via son constructeur.

Exemples

```
# Deux;;
- : numero = Deux

# let plusieurs n =
      n = Deux || n = Trois;;
val plusieurs : numero -> bool = <fun>
```

```
# plusieurs Un;;
- : bool = false

# plusieurs Trois;;
- : bool = true
```

Certains constructeurs d'un type somme peuvent avoir un argument. Un *argument* est une **valeur attachée à un constructeur**.

Ainsi, si `Id1`, ..., `Idn` sont des identificateurs, et `T` est un type,

```
Id1 | ...| Idk of T | ...| Idn
```

est un type somme dans lequel toute valeur `Idk` est attachée à une valeur de type `T`.

Exemple

Soit le type `nombre` défini par

```
type nombre = Entier of int | Rationnel of int * int | Infini
```

Une meilleure indentation :

```
type nombre =
    |Entier of int
    |Rationnel of int * int
    |Infini
```

Les constructeurs `Entier` et `Rationnel` possèdent des arguments.

Une valeur d'un type somme s'écrit par son constructeur suivi de son argument (s'il en a un).

Exemples

```
# Entier 13;;
- : nombre = Entier 13
```

```
# Rationnel (2, 3);;
- : nombre = Rationnel (2, 3)
```

```
# Infini;;
- : nombre = Infini
```

Une *liste d'entiers* est la liste vide **ou bien** une cellule contenant un entier qui est attachée à une liste d'entiers.

Cette définition se traduit en le type somme **récursif** ci-contre :

```
type liste_int =
    |Vide
    |Cellule of int * liste_int
```

Exemple

Nous construisons des listes d'entiers de la manière suivante :

```
# let e = Vide;;
val e : liste_int = Vide

# let e = Cellule (1, e);;
val e : liste_int = Cellule (1, Vide)

# let e = Cellule (2, e);;
val e : liste_int = Cellule (2, Cellule (1, Vide))

# let e = Cellule (3, e);;
val e : liste_int = Cellule (3, Cellule (2, Cellule (1, Vide)))
```

Le nom `e` est finalement lié à la liste $(3, 2, 1)$.

Un *arbre binaire d'entiers* est l'arbre vide **ou bien** un nœud contenant un entier qui est attaché à deux arbres binaires d'entiers.

Cette définition se traduit en le type somme **récursif** ci-contre :

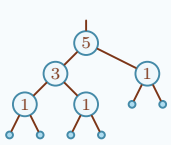
```
type arbre_b_int =
    |Vide
    |Noeud of arbre_b_int * int * arbre_b_int
```

Exemple

Nous construisons des arbres binaires d'entiers de la manière suivante :

```
# let a0 = Vide;;
val a0 : arbre_b_int = Vide
# let a1 = (Noeud (a0, 1, a0));;
val a1 : arbre_b_int = Noeud (Vide, 1, Vide)
# let a2 = (Noeud (a1, 3, a1));;
val a2 : arbre_b_int = Noeud (Noeud (Vide, 1, Vide), 3, Noeud (Vide, 1, Vide))
# let a3 = (Noeud (a2, 5, a1));;
val a3 : arbre_b_int = Noeud (Noeud (Noeud (Vide,1,Vide), 3, Noeud (Vide,1,Vide)), 5, Noeud (Vide,1,Vide))
```

Le nom `a3` est lié l'arbre binaire

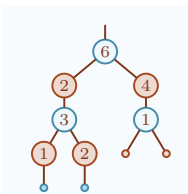


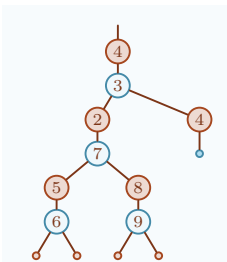
Un *arbre unaire binaire d'entiers* est

- l'arbre vide ou bien
- un nœud binaire contenant un entier, attaché à deux arbres unaires binaires dont les racines sont d'arités 1 ou bien
- un nœud unaire contenant un entier, attaché à un arbre unaire binaire dont la racine est d'arité 2.

Exemples

Voici deux arbres unaires binaires, respectivement dont la racine est d'arité 2 et dont la racine est d'arité 1 :





La définition de ces objets se traduit en les définitions de types suivantes.

Nous commençons par définir deux types pour représenter les arbres unaires binaires en séparant les cas en fonction de l'arité de la racine :

```
type arbre_1 =
    |Vide1
    |Noeud1 of int * arbre_2

and arbre_2 =
    |Vide2
    |Noeud2 of arbre_1 * int * arbre_1
```

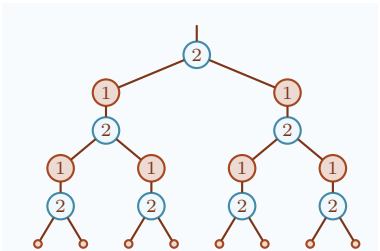
Le mot-clé `and` permet de réaliser des **définitions de types simultanées** : les types `arbre_1` et `arbre_2` sont en effet **mutuellement récursifs**.

Étant donné qu'un arbre unaire binaire peut être vide, avoir une racine unaire ou bien une racine binaire, nous définissons le type final par la somme (c'est-à-dire l'**union disjointe**) de ces trois possibilités :

```
type arbre_12 =
    |Vide
    |Arbre1 of arbre_1
    |Arbre2 of arbre_2
```

Exemple

Écrivons une expression de type `arbre_12` qui représente l'arbre




```
# let a = Vide1 in
  let a = Noeud2 (a, 2, a) in
  let a = Noeud1 (1, a) in
  let a = Noeud2 (a, 2, a) in
  let a = Noeud1 (1, a) in
  Arbre2 (Noeud2 (a, 2, a));;
```

```
- : arbre_12 =
Arbre2
 (Noeud2
   (Noeud1 (1,
     Noeud2 (Noeud1 (1, Noeud2 (Vide1, 2, Vide1)), 2,
      Noeud1 (1, Noeud2 (Vide1, 2, Vide1)))),
   2,
   Noeud1 (1,
    Noeud2 (Noeud1 (1, Noeud2 (Vide1, 2, Vide1)), 2,
     Noeud1 (1, Noeud2 (Vide1, 2, Vide1))))))
```

3.5.4. Types paramétrés

Tout comme les **fonctions** qui admettent des **paramètres** (voués à être substitués par des **valeurs**), il est possible de définir des **types** avec des **paramètres** (voués à être substitués par des **types**).

Il s'agit de *types paramétrés*.

La définition d'un nouveau type paramétré `ID` se fait par

```
type ('P1, ..., 'Pn) ID = OP
```

où

- `P1`, ..., `Pn` sont des identificateurs ;
- `OP` fait intervenir des types, des opérateurs de types et des paramètres `'P1`, ..., `'Pn`.

Les `'P1`, ..., `'Pn` sont des *paramètres de types*.

Lorsque `n = 1`, la définition se fait simplement (sans parenthèses) par

```
type 'P1 ID = OP
```

Supposons que `T` soit un type défini par

```
type ('P1, ..., 'Pn) T = OP
```

Nous disons alors que `T` *est paramétré* par `'P1`, ..., `'Pn`.

- Dans la définition de `T`, les occurrences de `'Pi`, $1 \leqslant i \leqslant n$, qui y figurent peuvent se penser comme étant **n'importe quel type**.

  Ainsi, un type paramétré désigne un **ensemble de types**.

- Si `T1`, ..., `Tn` sont des types, `(T1, ..., Tn) T` désigne le type dans lequel chaque `Pi` est **spécialisé** à `Ti`.

  Ainsi, `T` est un **opérateur de types**.

Ce mécanisme de définition de types avec des paramètres permet de construire des types faisant intervenir

1. des types scalaires (vus comme des constantes) ;
2. des paramètres de types (vus comme des variables) ;
3. des opérateurs de types.

Le type à un paramètre ci-contre permet de représenter des listes dont chaque cellule contient une valeur d'un type quelconque `e`. Nous obtiennons ainsi des **listes génériques**.

```
type 'e liste =
    |Vide
    |Cellule of 'e * 'e liste
```

Attention : les listes génériques sont **homogènes**, c'est-à-dire que les éléments de ses cellules sont d'un type quelconque mais tous d'un même type.

Exemples

- Une liste de caractères :

```
# Cellule ('a', (Cellule ('b', Vide)));;
- : char liste = Cellule ('a', Cellule ('b', Vide))
```

- Une liste de listes d'entiers :

```
# Cellule ((Cellule (1, Vide)), Vide);;
- : int liste liste = Cellule (Cellule (1, Vide), Vide)
```

- Une liste dont le type des éléments n'est pas déterminé :

```
# Vide;;
- : 'a liste = Vide
```

Le type à deux paramètres ci-contre permet de représenter des couples dont les coordonnées sont de types quelconques (et potentiellement différents).

```
# type ('t1, 't2) couple = {x : 't1 ; y : 't2};;
type ('a, 'b) couple = { x : 'a; y : 'b; }
```

Il ne faut pas confondre avec le type ci-contre où les deux coordonnées doivent être d'un même type.

```
# type 't couple_autre = {x : 't ; y : 't};;
type 't couple_autre = { x : 't; y : 't; }
```

L'interpréteur donne sa réponse en renommant les paramètres de types en `'a`, `'b`, ..., (prononcés habituellement « alpha », « beta », ...) selon leur ordre d'apparition.

Exemples

- Définition d'un couple :
```
# let c = {x = 4 ; y = 'd'};;
val c : (int, char) couple = {x = 4; y = 'd'}
```
- Création d'un autre couple basé sur `c` :
```
# {c with y = 'e'};;
- : (int, char) couple = {x = 4; y = 'e'}
```
- Création d'un autre couple basé sur `c` :
```
# {c with y = 2.2};;
- : (int, float) couple = {x = 4; y = 2.2}
```

Nous souhaitons définir un type pour représenter les **arbres unaires binaires** où

- les nœud unaires contiennent tous des valeurs d'un même type ;
- les nœuds binaires contiennent tous des couples dont les coordonnées peuvent être de types différents.

Nous avons ainsi besoin de trois paramètres de types : `'u` pour le type des valeurs des nœuds unaires, et `'x` et `'y` pour ceux des nœuds binaires.

```
type ('u, 'x, 'y) arbre_1 =
    |Vide1
    |Noeud1 of 'u * ('u, 'x, 'y) arbre_2

and ('u, 'x, 'y) arbre_2 =
    |Vide2
    |Noeud2 of ('u, 'x, 'y) arbre_1 * ('x * 'y) * ('u, 'x, 'y) arbre_1
```

```
type ('u, 'x, 'y) arbre_12 =
    |Vide
    |Arbre1 of ('u, 'x, 'y) arbre_1
    |Arbre2 of ('u, 'x, 'y) arbre_2
```

Nous cherchons à représenter des **mots** sur un alphabet quelconque.

Pour rappel, un *mot* est une suite finie de lettres qui appartiennent à un alphabet donné.

Les lettres d'un mot u sont indicées de 1 à sa longueur $|u|$ et pour tout $1 \leqslant i \leqslant |u|$, u_i désigne la i^{e} lettre de u.

Exemple

Le mot $u := \texttt{baacba}$ vérifie entre autres $u_1 = \texttt{b}$ et $u_4 = \texttt{c}$.

Usuellement, en **programmation impérative** (et donc dans un style **mutable**), un mot u est représenté par un tableau `tab` qui contient ses lettres ($u_i = \texttt{tab[i - 1]}$). Ceci supporte les opérations de lecture et modification de lettres, ainsi que de concaténation.

En **programmation fonctionnelle** (et donc dans un style **non mutable**), il est possible d'utiliser une liste simplement chaînée pour représenter les lettres du mot. Il est cependant possible d'adopter une approche beaucoup plus intéressante.

Idée principale : pour connaître un mot u, il suffit de connaître pour toute position i la lettre u_i de u.

Un mot est donc une **fonction** qui associe à chaque position i une lettre.

Nous obtenons donc le type produit nommé à un paramètre

```
type 'a mot = {
    lettres : int -> 'a;
    longueur : int
}
```

Quelques explications :

- le paramètre de type `'a` renseigne sur le type des lettres du mot ;
- le champ `lettres` est la fonction dont le rôle a été décrit plus haut ;
- le champ `longueur` contient la longueur du mot (ceci est nécessaire car on ne peut pas déduire la longueur du mot uniquement depuis la fonction `lettres` ; il serait possible d'utiliser un marqueur de fin de mot, mais cela rendrait le type plus compliqué).

Exemples

```
# let mot_1 = {
      lettres =
          (fun i ->
              if i = 2 || i = 3 then
                  'a'
              else
                  'b'
          );
      longueur = 5
};;
val mot_1 : char mot
    = {lettres = <fun>; longueur = 5}
```

Ceci lie au nom `mot_1` le mot de caractères `baabb`.

Le champ `lettres` est une fonction qui envoie `1` sur `'b'`, `2` sur `'a'`, `3` sur `'a'`, `4` sur `'b'` et `5` sur `'b'`.

Notons la mise entre parenthèses de la fonction anonyme définissant le champ `lettres`, obligatoire syntaxiquement ici.

```
# let mot_2 = {
      lettres = (fun i -> i mod 2 = 0);
      longueur = 4294967296
};;
val mot_2 : bool mot = {lettres = <fun>;
    longueur = 4294967296}
```

Ceci lie au nom `mot_2` le mot de booléens `false true false true ...` de longueur $4294967296 = 2^{32}$.

La quantité de mémoire utilisée est négligeable devant la longueur du mot.

Cette idée de représenter les mots par des fonctions peut s'appliquer à la **représentation d'images**.

Une *image* (dans le plan) peut se voir comme un tableau à deux dimensions de pixels.

Mieux : une image peut aussi se voir comme une **fonction** qui associe à chaque point une couleur.
Ceci permet de représenter des images de très hautes définitions avec peu de mémoire.

On obtient les types

```
type point = int * int
```

et

```
type 'a image = {
    contenus_pixels : point -> 'a;
    largeur : int;
    hauteur : int
}
```

Avec de plus la définition du type

```
type couleur = {rouge : int; vert : int; bleu : int}
```

toute valeur de type `couleur image` fait le travail.

Exemple

```
# let im_1 = {
      contenus_pixels =
          (fun p ->
              let (x, y) = p in
              if x = y then
                  {rouge = 127; vert = 127; bleu = 127}
              else
                  {rouge = 0; vert = 0; bleu = 0}
          );
      largeur = 16;
      hauteur = 16
};;
val im_1 : couleur image = {contenus_pixels = <fun>;
    largeur = 16; hauteur = 16}
```

Ceci lie au nom `im_1` une image de définition 16×16 dont les pixels sur la diagonale sont gris et les autres noirs.

Exemple

```
# let im_2 = {
      contenus_pixels =
          (fun p ->
              let (x, y) = p in
              if (sqrt ((float_of_int x) ** 2. +. (float_of_int y) ** 2.)) <= 1024. then
                  {rouge = 0; vert = 0; bleu = 0}
              else
                  {rouge = 255; vert = 255; bleu = 255}
          );
      largeur = 1048576;
      hauteur = 1048576
}
val im_2 : couleur image = {contenus_pixels = <fun>; largeur = 1048576; hauteur = 1048576}
```

Ceci lie au nom `im_2` une image (de très haute définition) d'un disque noir sur un fond blanc.

Des manipulations (simples) de ces images figureront comme exemples de certains concepts apparaissant plus loin dans ce cours.

3.6. Récursivité

3.6.1. Définition de fonctions récursives

Pour réaliser des **définitions récursives** (c'est-à-diire lier des valeurs à un nom en faisant référence au nom lui-même), nous utilisons la construction

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

où `ID` est un identificateur, `P1`, ..., `Pn` sont ses paramètres et `EXP` est une expression.

Exemples

```
# let fact n =
      if n <= 1 then
          1
      else
          n * (fact (n - 1));;
Error: Unbound value fact
```

Cette définition sans le `rec` pose problème : l'identificateur `fact` n'est lié à aucune valeur lorsque l'on fait appel à sa valeur en l. 5.

```
# let rec fact n =
      if n <= 1 then
          1
      else
          n * (fact (n - 1));;
val fact : int -> int = <fun>
```

Ceci définit bien la fonction factorielle.

```
# fact 7;;
- : int = 5040
```

Lors de la liaison d'un nom `s` à une valeur en utilisant `rec`, toute occurrence de `s` qui figure dans sa définition fait référence au nom `s` qui est en train d'être défini.

Ainsi, le `rec` change la portée de `s`.

Exemples

Comparons les expressions suivantes :

```
# let x = 10 in
      let x = 20 in
          x + x;;
Warning 26: unused variable x.
- : int = 40
```

```
# let x = 10 in
      (let x = 20 in
          x)
      + x;;
- : int = 30
```

```
# let rec x =
      let x = 20 in
          x + x;;
val x : int = 40
```

```
# let rec x =
      (let x = 20 in
          x)
      + x;;
Error: This kind of expression is not
allowed as right-hand side of 'let rec'
```

Exemples

```
# let f x =
      let f y =
          y + (f (x - 1))
      in
      if x = 0 then
          0
      else
          x + (f (x - 1));;
Error: Unbound value f
```

```
# let f x =
      let rec f y =
          y + (f (x - 1))
      in
      if x = 0 then
          0
      else
          x + (f (x - 1));;
val f : int -> int = <fun>
# f 3;;
Stack overflow...
```

```
# let rec f x =
      let f y =
          y + (f (x - 1))
      in
      if x = 0 then
          0
      else
          x + (f (x - 1));;
val f : int -> int = <fun>
# f 3;;
- : int = 9
```

```
# let rec f x =
     let rec f y =
         y + (f (x - 1))
     in
     if x = 0 then
         0
      else
          x + (f (x - 1));;
val f : int -> int = <fun>
# f 3;;
Stack overflow...
```

La construction `let rec` est compatible avec les **liaisons simultanées** (construction `let ... and ...`).

Il est ainsi possible de définir des **fonctions mutuellement récursives**.

Exemple

```
# let rec zero x =
      if x = 0 then
          "zero"
       else
           un (x - 1)
  and un x =
      if x = 0 then
          "un"
      else
          deux (x - 1)
  and deux x =
      if x = 0 then
          "deux"
      else
          zero (x - 1);;
val zero : int -> string = <fun>
val un : int -> string = <fun>
val deux : int -> string = <fun>
```

Ceci définit **trois fonctions mutuellement récursives**, simultanément.

L'appel `zero n` renvoie la chaîne de caractères renseignant sur le reste de la division de `n` par `3`.

Suite d'appels pour le calcul de `zero 4` :

`zero 4` → `un 3` → `deux 2` → `zero 1` → `un 0` → `"un"`.

3.6.2. Émulation des instructions de boucles

Il est possible de simuler les **instructions de boucle** propres au paradigme de programmation impérative à l'aide de **fonctions récursives locales**.

En effet, l'effet d'une suite d'instructions (en pseudo-code) utilisant une boucle « tant que » se traduit au moyen

1. d'une définition d'une fonction récursive utilisant une conditionnelle ;
2. d'un appel à cette fonction.

Instruction de boucle :

```
Tant que C :
    I
Fin
```

Équivalent fonctionnel :

```
Fonction rec f :
    Si C :
        I
        Appel à f
    Fin
Fin
Appel à f
```

Ici, C est une condition et I est une expression.

Exemple --- boucle `while` simple

Fonction en C :

```c
int triangle(int n) {
    int i, res;

    res = 0;
    i = n;
    while (i >= 1) {
        res += i;
        i -= 1;
    }

    return res;
}
```

Fonction en OCaml :

```ocaml
let triangle n =

    (* Fonction locale auxiliaire. *)
    let rec aux i =
        if i >= 1 then
            i + (aux (i - 1))
        else
            0
    in

    (* Appel à la fonction locale auxiliaire. *)
    aux n
```

Exemple --- boucle `for` simple

Fonction en C :

```c
int somme_pairs(int n) {
    int i, res;

    res = 0;
    for (i = 0 ; i <= n ; i += 2) {
        res += i;
    }

    return res;
}
```

Fonction en OCaml :

```
let somme_pairs n =

    (* Fonction locale auxiliaire. *)
    let rec aux i =
        if i <= n then
            i + (aux (i + 2))
        else
            0
    in

    (* Appel à la fonction locale auxiliaire. *)
    aux 0
```

3.6.3. Récursivité terminale

Considérons la fonction

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

et l'appel

```
fact 4
```

Cette dernière expression s'**évalue** au fil des appels récursifs en

```
fact 4 → 4 * (fact 3) → 4 * 3 * (fact 2) → 4 * 3 * 2 * (fact 1) → 4 * 3 * 2 * 1 → 24
```

Ce calcul, pour être mené à bien, a dû **garder en mémoire l'expression**

```
4 * 3 * 2 * 1
```

qui fait intervenir quatre (`n`) opérandes et trois (`n - 1`) opérateurs.

Considérons la fonction

```
let rec fact n acc =
    if n <= 1 then
        acc
    else
        fact (n - 1) (n * acc)
```

et l'appel

```
fact 4 1
```

Cette dernière expression s'évalue au fil des appels récursifs en

```
fact 4 1 → fact 3 (4 * 1) → fact 3 4 → fact 2 (3 * 4)
  → fact 2 12 → fact 1 (2 * 12) → fact 1 24 → 24
```

Ce calcul, pour être mené à bien, a dû garder en mémoire des expressions faisant intervenir au plus deux opérandes et un opérateur, en plus de l'appel de fonction.

La 2e version de la fonction `fact` possède la propriété d'être *récursive terminale* : le résultat de son appel récursif est renvoyé tel quel, sans l'adjoindre d'une opération.

En effet, dans la 1re version, la valeur de retour subit une multiplication

```
n * (fact (n - 1))
```

alors que dans la 2e, elle ne subit aucune modification

```
fact (n - 1) (n * acc)
```

Le calcul de la 1re version nécessite de garder en mémoire une expression de taille $\Theta(n)$, alors que celui de la 2e ne travaille que sur une expression de taille $\Theta(1)$.

Les fonctions récursives terminales utilisent **moins de mémoire** que leurs analogues non récursives terminales.

Quelques remarques :

- lorsque la version non récursive terminale d'une fonction n'est pas beaucoup plus simple que sa version récursive terminale, il est important de préférer la 2e version ;
- en revanche, si l'écriture récursive terminale d'une fonction dénature l'esprit d'un algorithme, il est intéressant de proposer une version non récursive terminale.

  C'est souvent le cas lors de l'écriture de fonction manipulant des **structures arborescentes**.

Pour écrire des fonctions récursives terminales, on utilise un paramètre dont le rôle est de contenir le résultat au fur et à mesure des appels. C'est l'*accumulateur*.

Il faut appeler la fonction avec une bonne valeur de départ pour l'accumulateur (en général cette valeur correspond au **cas terminal de la récursion**).

En général, il est d'usage d'**enrober** une fonction avec accumulateur pour la rendre plus facilement utilisable (ceci l'appelle avec la bonne valeur initiale pour l'accumulateur).

Exemple

Version non enrobée :

```
let rec fact n acc =
    if n <= 1 then
        acc
    else
        fact (n - 1) (n * acc)
```

Version enrobée :

```
let fact n =
    let rec aux n acc =
        if n <= 1 then
            acc
        else
            aux (n - 1) (n * acc)
    in
    aux n 1
```

Exemple

```
let rec fibo n =
    if n <= 1 then
        n
    else
        (fibo (n - 1)) + (fibo (n - 2))
```

C'est la version non récursive terminale de la fonction calculant le `n`e nombre de Fibonacci.

En effet, l'appel récursif (double) est adjoint d'une opération (somme).

```
let fibo n =
    let rec aux n acc1 acc2 =
        if n = 0 then
            acc2
        else if n = 1 then
            acc1
        else
            aux (n - 1) (acc1 + acc2) acc1
    in
    aux n 1 0
```

C'est la version récursive terminale de la fonction précédente.

Elle utilise deux accumulateurs (à cause du double appel récursif précédent).

`acc1` contient la valeur du `n - 1`e nombre de Fibonacci et `acc2` contient la valeur du `n - 2`e nombre de Fibonacci.

Une fonction récursive terminale F a pour *forme générale*

```
Fonction récursive F(x_1,...,x_n, a_1,...,a_m) :
    Si C(x_1,...,x_n) :
        F(φ(x_1,...,x_n), ψ(a_1,...,a_m))
    Sinon :
        R(a_1,...,a_m)
    Fin
Fin
```

où

- $x_1, \ldots, x_n$ sont les paramètres (**entrées**) ;
- $a_1, \ldots, a_m$ sont les accumulateurs (**sorties**) ;
- $C(x_1, \ldots, x_n)$ est une **expression booléenne** dépendant des entrées ;
- $\phi(x_1, \ldots, x_n)$ renvoie le n-uplet **préparant les entrées** pour l'appel récursif ;
- $\psi(a_1, \ldots, a_m)$ renvoie le m-uplet **préparant les accumulateurs** pour l'appel récursif ;
- $R(a_1, \ldots, a_m)$ est une **expression résultat** obtenue à partir des accumulateurs.

La *dérécursivation* est un procédé qui permet de transformer toute fonction récursive terminale en une fonction itérative.

Voici une fonction récursive terminale dans sa forme générale F et sa version dérécursivée G :

```
Fonction récursive F(x_1,...,x_n, a_1,...,a_m) :
    Si C(x_1,...,x_n) :
        F(φ(x_1,...,x_n), ψ(a_1,...,a_m))
    Sinon :
        R(a_1,...,a_m)
    Fin
Fin
```

```
Fonction itérative G(x_1,...,x_n, a_1,...,a_m) :
    Tant que C(x_1,...,x_n) :
        (x_1,...,x_n) := φ(x_1,...,x_n)
        (a_1,...,a_m) := ψ(a_1,...,a_m)
    Fin
    Renvoyer R(a_1,...,a_m)
Fin
```

Les notations sont ici les mêmes que celles utilisées précédemment.

Note : ceci est une forme plus générale et précise de ce qui a été vu concernant l'émulation des instructions de boucles par des fonctions récursives.

Exemple

Fonction en forme récursive habituelle :

```
let fibo n =
    let rec aux n acc1 acc2 =
        if n = 0 then acc2
        else if n = 1 then acc1
        else
            aux (n - 1) (acc1 + acc2) acc1
    in
    aux n 1 0
```

Fonction en forme générale :

```
let rec fibo n acc1 acc2 =
    if n >= 2 then
        fibo (n - 1) (acc1 + acc2) acc1
    else if n = 0 then
        acc2
    else
        acc1
```

Version dérécursivée en C :

```
int fibo(int n, int acc1, int acc2) {
    while (n >= 2) {
        n = n - 1;
        acc1 = acc1 + acc2;
        acc2 = acc1 - acc2;
    }
    if (n == 0) return acc2;
    else return acc1;
}
```

3.7. Filtrage de motifs

3.7.1. Principes généraux

Considérons le type

```
type point =
    |Droite of int
    |Plan of int * int
    |Espace of int * int * int
```

Nous souhaitons écrire une fonction `oppose` de type `point -> point` qui renvoie l'opposé du point argument (obtenu en changeant le signe de ses coordonnées).

Une très bonne manière de faire consiste à utiliser un **filtrage de motifs** :

```
let oppose p =
    match p with
        |Droite x -> Droite (-x)
        |Plan (x, y) -> Plan (-x, -y)
        |Espace (x, y, z) -> Espace (-x, -y, -z)
```

Exemples

```
# oppose (Plan (3, 1));;
- : point = Plan (-3, -1)
```

```
# oppose (Espace(1, 0, -1));;
- : point = Espace (-1, 0, 1)
```

La construction syntaxique

```
match EXP with
    |MOTIF1 -> EXP1
    ...
    |MOTIFn -> EXPn
```

où `EXP`, `EXP1`, ..., `EXPn` sont des expressions toutes d'un même type et `MOTIF1`, ..., `MOTIFn` des motifs, met en place un *filtrage de motifs* sur `EXP`.

Chaque ligne `MOTIFi -> EXPi` est une *clause*.

L'évaluation de cette expression se déroule comme suit :

1. `EXP` est évaluée ;
2. on essaye de *filtrer* (faire correspondre) la valeur de `EXP` avec l'un des motifs, de haut en bas ;
3. si un motif `MOTIFi` filtre la valeur de `EXP`, la **valeur** de toute l'expression est celle de `EXPi` ;
4. si aucun motif ne filtre la valeur de `EXP`, une **erreur** est signalée dynamiquement (à l'exécution).

Le filtrage de motifs peut se penser en 1re approximation comme le `switch` du C. Il est dans les faits **beaucoup plus puissant**.

Il s'utilise principalement :

1. lorsque le traitement à effectuer dépend davantage de la **structure d'une valeur** que de la valeur elle-même.

Exemple

```
let dimension p =
    match p with
        |Droite x -> 1
        |Plan (x, y) -> 2
        |Espace (x, y, z) -> 3
```

renvoie le nombre de coordonnées du point `p`.

2. lorsque l'on souhaite d'accéder à une partie d'une valeur, le filtrage permettant de **déconstruire**.

Exemple

```
let projection_x p =
    match p with
        |Droite x -> x
        |Plan (x, y) -> x
        |Espace (x, y, z) -> x
```

renvoie la première coordonnée du point `p`.

L'interpréteur vérifie si le filtrage est *exhaustif*, c'est-à-dire si toute expression du type attendu **peut être filtrée par au moins un motif**.

Si ça n'est pas le cas, un avertissement est signalé.

Comme expliqué précédemment, ceci peut provoquer des erreurs à l'exécution.

Exemple

```
# let dimension p =
      match p with
          |Droite x -> 1
          |Espace (x, y, z) -> 3;;
```

```
Warning 8: ... not exhaustive. ...
value that is not matched: Plan (_, _)
val dimension : point -> int = <fun>
```

Le joker `_` est un motif universel : il filtre toute valeur. Son utilisation rend donc tous les filtrages exhaustifs.

Exemple

```
# let entier_vers_chaine n =
      match n with
          |0 -> "zero"
          |1 -> "un"
          |2 -> "deux"
          |_ -> "autre";;
```

```
val entier_vers_chaine :
    int -> string = <fun>
```

Il est possible de **raffiner le filtrage** en incluant des tests à des clauses

```
MOTIF -> EXP
```

Nous utilisons pour cela la syntaxe

```
MOTIF when TEST -> EXP
```

où `TEST` est une expression de type `bool` appelée *garde*.

Pour que ce motif filtre une expression, il faut en plus que la valeur de `TEST` soit `true`.

Exemple

```
let est_dans_quart_de_plan p =
    match p with
        |Droite _ -> false
        |Plan (x, y) when x >= 0 && y >= 0 -> true
        |Plan (_, _) (* ou meme 'Plan _' *) -> false
        |Espace (_, _, _) (* ou meme 'Espace _' *) -> false
```

Ceci teste si l'argument est un point du plan à coordonnées positives.

Il existe trois sortes de motifs :

1. les motifs **constants**, qui sont des constantes habituelles (`0`, `true`, `()`, `'a'`, `"abc"`, *etc.*) ;
2. les motifs **à paramètre**, qui sont des motifs qui utilisent des noms ou bien des jokers ;
3. les motifs **composés**, qui sont des motifs qui utilisent des constructeurs ou des enregistrements.

La considération d'une clause peut réaliser des **liaisons locales** : ici, la considération des motifs lie localement `x`, `y` et/ou `z` à des valeurs.

Exemple

```
let somme p =
    match p with
        |Droite x -> x
        |Plan (x, y) -> x + y
        |Espace (x, y, z) -> x + y + z
```

Les noms dans les motifs ne font pas référence à leur définition passée.

Exemple

```
let f x =
    let n = 2 in
    match x with
        |0 -> 0
        |n -> -1
        |_ -> 1
```

```
# f 0;;
- : int = 0

# f 3;;
- : int = -1
```

Le `n` du 2e motif ne fait pas référence à la liaison précédente.

Ce motif `n` **filtre tout**.

3.7.2. Formules logiques

Nous souhaitons représenter des **formules du calcul des prédicats** et définir une fonction qui permet d'**évaluer une formule** sous une valuation donnée.

Une *formule* est une donnée récursive :

1. il s'agit d'un atome p ;
2. ou bien de la négation d'une formule $(\neg F)$;
3. ou bien de la conjonction de deux formules $(F_1 \wedge F_2)$;
4. ou bien de la disjonction de deux formules $(F_1 \vee F_2)$.

Ceci se traduit en le type récursif

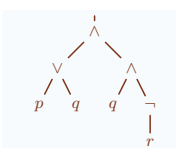
```
type formules =
    |Atome of char
    |Non of formules
    |Et of formules * formules
    |Ou of formules * formules
```

Exemple

La formule logique $(p \vee q) \wedge (q \wedge (\neg r))$ est représentée par l'expression

```
Et (Ou ((Atome 'p'), (Atome 'q') ), Et ((Atome 'q'), (Non (Atome 'r'))))
```

Il s'agit également de l'arbre



Nous souhaitons maintenant écrire une fonction permettant d'**évaluer** une formule.

Évaluer une formule se fait toujours dans le contexte d'une *valuation* donnée : elle informe sur la valeur de vérité (« vrai » ou « faux ») prise par chacun des atomes de la formule. Ces valeurs de vérité sont représentées par le type

```
type valeurs_verite = Faux | Vrai
```

Exemple

Une valuation possible (parmi d'autres) de la formule $(p \vee q) \wedge (q \wedge (\neg r))$ est

$$p \mapsto \text{vrai}, \quad q \mapsto \text{faux}, \quad r \mapsto \text{vrai}.$$

Il y en a $2^3 = 8$ différentes car chaque atome peut prendre deux valeurs de vérité différentes.

Un type possible pour représenter une valuation est la **liste associative**

```
type valuations_assoc_list = (char * valeurs_verite) list
```

Exemple

La valuation précédente est représentée par la liste associative

```
[('p', Vrai); ('q', Faux); ('r', Vrai)]
```

Une valuation se représente élégamment par une **fonction** attribuant à un atome sa valeur de vérité

```
type valuations = char -> valeurs_verite
```

Exemple

La valuation précédente est représentée par la fonction ci-contre.

Celle-ci n'est cependant pas correcte car le filtrage de motifs y serait incomplet.

```
let valu a =
    match a with
        |'p' -> Vrai
        |'q' -> Faux
        |'r' -> Vrai
```

Ce filtrage se complète en **levant une exception** dans les cas problématiques via `raise`, fonction de type `exn -> 'a`, où `exn` est le type des exceptions. Une exception `EX` se définit via `exception EX`.

Exemple

La valuation précédente devient correcte ainsi.

Une clause joker est ajoutée. Celle-ci lève l'exception `Erreur` définie en premier lieu.

Notons la factorisation des cas similaires pour `'p'` et `'r'`.

```
exception Erreur
let valu a =
    match a with
        |'p' |'r' -> Vrai
        |'q' -> Faux
        |_ -> raise Erreur
```

Nous pouvons enfin mettre en place la **fonction d'évaluation**.

Elle s'écrit très simplement au moyen d'un filtrage :

```
let rec evaluer valu form =
    match form with
        |Atome a -> valu a
        |Non f -> if evaluer valu f = Faux then Vrai else Faux
        |Et (f1, f2) -> if evaluer valu f1 = Vrai && evaluer valu f2 = Vrai then Vrai else Faux
        |Ou (f1, f2) -> if evaluer valu f1 = Faux && evaluer valu f2 = Faux then Faux else Vrai
```

Exemple

En considérant la formule précédente $(p \vee q) \wedge (q \wedge (\neg r))$ liée au non `f` et sa valuation précédente

$$p \mapsto \text{vrai}, \quad q \mapsto \text{faux}, \quad r \mapsto \text{vrai},$$

liée au nom `valu`, nous l'évaluons par

```
match evaluer valu f with
    |Vrai -> print_endline "Vrai"
    |Faux -> print_endline "Faux"
```

Ceci imprime `Faux`.

3.7.3. Type option

Une *valeur optionnelle* de type `T` est

- soit une **valeur spéciale** ;
- soit une valeur de type `T` **encapsulée**.

Elles permettent d'écrire des fonctions qui, en cas d'échec, renvoient la valeur spéciale.

Ceci est implanté dans la bibliothèque standard par le type paramétré `'a option`, dont la définition est

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

Exemple

Pour éviter les erreurs de division par `0`, nous mettons en place la **division sûre** par

```
# let division_sure a b =
    if b = 0 then
        None
    else
        Some (a / b)
val division_sure : int -> int -> int option = <fun>
```

```
# division_sure 32 0;;
- : 'a option = None

# division_sure 32 4;;
- : int option = Some 8
```

Les fonctions à type de retour optionnel offrent une **alternative fonctionnelle au mécanisme des exceptions**, qui ont le désavantage de ne pas apparaître dans les types des fonctions.

La fonction `Option.get` de type `'a option -> 'a` permet de **renvoyer la valeur encapsulée** par une valeur optionnelle. Une exception est levée lorsque cette fonction est appliquée à `None`.

Exemples

```
# Option.get (division_sure 32 4);;
- : int = 8
```

```
# Option.get (division_sure 32 0);;
Exception: Invalid_argument "option is None".
```

Les fonctions `Option.is_some` et `Option.is_none` de type `'a option -> bool` **testent** respectivement si une valeur optionnelle encapsule une valeur ou si il s'agit de la valeur spéciale.

Exemples

```
# Option.is_some (Some "abc");;
- : bool = true
```

```
# Option.is_none (Some "abc");;
- : bool = false
```

La fonction `Option.map` de type `('a -> 'b) -> 'a option -> 'b option` est telle que `Option.map f opt` renvoie `Some (f v)` si `opt` encapsule la valeur `v` et `None` sinon.

Exemples

```
# Option.map succ (Some 1);;
- : int option = Some 2
```

```
# Option.map succ None;;
- : int option = None
```

La fonction `Option.bind` de type `'a option -> ('a -> 'b option) -> 'b option` est telle que `Option.bind opt f` renvoie `f v` si `opt` encapsule la valeur `v` et `None` sinon.

Exemples

```
# Option.bind (Some 1) (fun n -> Some (succ n));;
- : int option = Some 2
```

```
# Option.bind None (fun n -> Some (succ n));;
- : int option = None
```

Revenons à la fonction `Option.get`. Elle possède comme défaut majeur de **lever une exception** lorsqu'elle est appliquée sur `None`.

Une fonction importante à considérer est

```
# let get_default opt def =
      match opt with
          |None -> def
          |Some v -> v
val get_default : 'a option -> 'a -> 'a = <fun>
```

Elle est telle que `get_default opt def` renvoie la valeur encapsulée par `opt` si elle existe et renvoie `def` sinon.

Exemple

Dans le contexte de la représentation des formules logiques vu précédemment, une valuation peut se représenter par une valeur de vérité encapsulée. Ainsi,

```
exception Erreur
let valu a =
    match a with
        |'p' |'r' -> Vrai
        |'q' -> Faux
        |_ -> raise Erreur
```

devient

```
let valu a =
    match a with
        |'p' |'r' -> Some Vrai
        |'q' -> Some Faux
        |_ -> None
```

L'ancien code se met facilement à jour en utilisant les fonctions précédentes du module `Option`.

3.8. Fonctions d'ordre supérieur

3.8.1. Principes généraux

Une *fonction d'ordre supérieur* est une fonction `f` qui vérifie au moins l'une des deux conditions suivantes :

1. `f` possède un **paramètre** de type fonction ;
2. `f` **renvoie** une fonction.

Le fait de pouvoir paramétrer une fonction par une fonction permet d'avoir du code potentiellement **générique**.

Le fait de pouvoir renvoyer une fonction est un procédé très puissant en programmation fonctionnelle. Le programmeur n'est plus le seul concepteur de fonctions : l'exécution peut en **créer à la volée** et en appeler.

Rappel : étant donné que les fonctions sont conceptualisées de manière curryfiée, une fonction à $n \geqslant 2$ entrées est une fonction à une entrée qui **renvoie une fonction** à $n - 1$ entrées. Ainsi, toute fonction à $n \geqslant 2$ entrées est elle-même déjà une fonction d'ordre supérieur.

Exemple

Analysons le type de la fonction

```
let rec appli_repetee f x n =
    if n = 0 then
        x + 0
    else
        f (appli_repetee f x (n - 1))
```

Nous inférons le type

```
(int -> int) -> int -> int -> int
```

qui montre que `appli_repetee` est paramétrée par une fonction `int -> int`. C'est donc une fonction d'ordre supérieur.

Elle calcule l'application de la composée n^e de `f` sur l'entier `x`, c'est-à-dire,

$$f^n(x)$$

Par exemple

```
# appli_repetee (fun x -> x + 1) 3 4;;
- : int = 7
```

```
# appli_repetee (fun x -> 2 * x) 3 4;;
- : int = 48
```

Exemple

Analysons le type de la fonction

```
let compose f1 f2 =
    fun x -> f1 (f2 x)
```

Nous inférons le type

```
('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
```

qui montre que `compose` est paramétrée par deux fonctions `f1` et `f1`, de types respectifs `'a -> 'b` et `'c -> 'a`. C'est donc une fonction d'ordre supérieur.

Elle calcule la fonction définie comme la composée de `f1` avec `f2`, c'est-à-dire,

$$f1 \circ f2$$

Par exemple,

```
# let f = compose succ succ;;
val f : int -> int = <fun>

# f 0;;
- : int = 2
```

```
# compose int_of_float Float.round 2.5001;;
- : int = 3
```

L'opérateur d'*application inversée* est l'opérateur binaire `|>`. Il est de type

```
'a -> ('a -> 'b) -> 'b
```

Il permet, étant données une fonction `f` de type `'a -> 'b` et une expression `e` de type `'a`, d'écrire

```
e |> f
```

à la place de

```
f e
```

Il sert à rendre les **applications successives de fonctions plus naturelles**.

En effet, si `e` est une expression de type `'a0` et chaque f_i est une fonction de type $'a_{i-1}$ `->` $'a_i$, il devient possible d'écrire

```
e |> f1 |> f2 |> ... |> fn
```

à la place de

```
fn (... (f2 (f1 e)) ...)
```

Remarque : pour bien s'articuler avec cet opérateur, il est intéressant d'écrire les fonctions avec le paramètre principal en dernière position.

3.8.2. Mots fonctionnels

Considérons la façon fonctionnelle de représenter les mots vue précédemment via le type

```
type 'a mot = {
    lettres : int -> 'a;
    longueur : int
}
```

Rappelons que si `u` est un mot, l'expression `u.lettres i` a pour valeur la i^e lettre de `u`.

Exemple

```
let mot_3 = {
    lettres =
    (fun i ->
        match i with
            |1 -> 'a'
            |2 -> 'b'
            |_ -> 'b'
    );
    longueur = 3
}
```

Ceci représente représente le mot `abb` dont les lettres de type `char`.

Remarque : il est encore meilleur de faire en sorte que le champ `lettres` soit de type `int -> 'a option` afin de renvoyer `None` dans le cas où on essaye de lire une lettre à une position non autorisée. Ceci n'est pas illustré ici.

La fonction d'ordre supérieur ci-contre renvoie la chaîne de caractères qui représente le mot `u` :

```
let vers_chaine lettre_vers_char u =
    let rec aux i =
        if i = u.longueur + 1 then
            ""
        else
            let ch = String.make 1 (lettre_vers_char (u.lettres i)) in
            ch ^ (aux (i + 1))
    in
    aux 1
```

Quelques explications :

- cette fonction d'ordre supérieur est de type `('a -> char) -> 'a mot -> string` ;
- `lettre_vers_char` est une fonction qui renvoie le caractère associé à toute lettre de type `'a` ;
- l'expression `String.make n c` est la chaîne de caractères de longueur `n` faite de caractères `c`.

Exemples

```
# vers_chaine (fun x -> x) mot_3;;
- : string = "abb"
```

```
# vers_chaine
      (fun b -> if b then "T" else "F")
      {lettres = (fun _ -> true); longueur = 4};;
- : string = "TTTT"
```

La fonction ci-contre renvoie le mot défini comme étant la **concaténation** des deux mots `u` et `v` :

```
let concatener u v =
    let lettres i =
        if i <= u.longueur then
            u.lettres i
        else
            v.lettres (i - u.longueur)
    in
    {lettres = lettres ; longueur = u.longueur + v.longueur}
```

Quelques explications :

- cette fonction est de type `'a mot -> 'a mot -> 'a mot` ;
- pour construire le mot résultat, une fonction locale `lettres` est construite à partir des fonctions `u.lettres` et `v.lettres`. Elle se base sur le fait que $(uv)_i = u_i$ si $1 \leqslant i \leqslant |u|$ et $(uv)_i = v_{i-|u|}$ sinon.

Exemple

```
# let mot_4 = concatener mot_3 mot_3 in
  vers_chaine (fun x -> x) mot_4;;
- : string = "abbabb"
```

Nous pouvons aussi définir des mots particuliers, comme les **mots de Fibonacci**.

Pour tout $n \geqslant 0$, le n^{e} mot de Fibonacci f_n est défini par

$$f_n := \begin{cases} b & \text{si } n = 0, \\ a & \text{si } n = 1, \\ f_{n-1} f_{n-2} & \text{sinon.} \end{cases}$$

```
let rec mot_fibo n =
    if n = 0 then
        {lettres = (fun _ -> 'b'); longueur = 1}
    else if n = 1 then
        {lettres = (fun _ -> 'a'); longueur = 1}
    else
        concatener (mot_fibo (n - 1)) (mot_fibo (n - 2))
```

Exemples

Avec la liaison préalable `let vers_chaine_char = vers_chaine (fun x -> x)` qui, au passage, est une application partielle,

```
# let w = mot_fibo 0 in vers_chaine_char w;;
- : string = "b"

# let w = mot_fibo 1 in vers_chaine_char w;;
- : string = "a"

# let w = mot_fibo 2 in vers_chaine_char w;;
- : string = "ab"
```

```
# let w = mot_fibo 3 in vers_chaine_char w;;
- : string = "aba"

# let w = mot_fibo 4 in vers_chaine_char w;;
- : string = "abaab"

# let w = mot_fibo 5 in vers_chaine_char w;;
- : string = "abaababa"
```

3.8.3. Images fonctionnelles

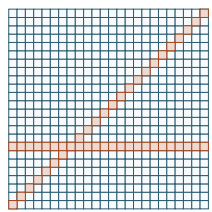
Considérons la façon fonctionnelle de représenter les images vue précédemment via les types ci-contre :

```
type points = int * int
type 'a images = {
    contenus_pixels : points -> 'a;
    largeur : int;
    hauteur : int
}
type couleurs = {rouge : int; vert : int; bleu : int}
```

Exemple

```
let im_3 = {
    contenus_pixels =
    (fun p ->
        let (x, y) = p in
        if x = y || y = 7 then
            {rouge = 127; vert = 127; bleu = 127}
        else
            {rouge = 255; vert = 255; bleu = 255}
    );
    largeur = 24;
    hauteur = 24
}
```

Le nom `im_3` est lié à la valeur de type `couleurs images` qui représente l'image (dans le plan cartésien positif)



Remarque : ici aussi, le type `points -> 'a option` pour le champ `contenus_pixels` est une excellente alternative.

La fonction suivante permet d'ouvrir une fenêtre graphique et de dessiner l'image `im`.

```
let dessiner im =
    Graphics.open_graph (Printf.sprintf " %dx%d" im.largeur im.hauteur);
    List.init im.largeur
        (fun x ->
            List.init im.hauteur
                (fun y ->
                    let c = im.contenus_pixels (x, y) in
                    let c' = Graphics.rgb c.rouge c.vert c.bleu in
                    (x, y, c')))
    |> List.flatten
    |> List.iter (fun (x, y, c) -> Graphics.set_color c; Graphics.plot x y);
    Graphics.wait_next_event [Button_down; Key_pressed] |> ignore;
    Graphics.close_graph ()
```

Quelques explications :

- cette fonction utilise le module `Graphics` permettant d'ouvrir des fenêtres graphiques, de dessiner à l'intérieur et de gérer les entrées sur clavier et souris ;
- la liste des triplets contenant les coordonnées des pixels et leur couleur est construite. Ensuite, cette liste est parcourue pour afficher chaque pixel ;
- une fois l'image dessinée, la fenêtre reste ouverte jusqu'à ce que l'utilisateur clique ou appuie sur une touche.

Voici des fonctions pour calculer l'image obtenue en **complémentant les couleurs** des pixels d'une image `im` :

```
let complementer_couleur c =
    let cpl x = 255 - x in
    {rouge = cpl c.rouge; vert = cpl c.vert; bleu = cpl c.bleu}

let complementer im =
    let contenus_pixels p =
        complementer_couleur (im.contenus_pixels p)
    in
    {im with contenus_pixels = contenus_pixels}
```

Tout appel `complementer im` s'évalue en temps $\Theta(1)$. Les complexités en temps et en espace ne dépendent pas de la taille de l'image.

En revanche, pour effectivement dessiner l'image, il n'est pas possible de faire autrement que de considérer tous les pixels et de les afficher un par un. Dans ce cas, la complexité en temps pour afficher le complémentaire d'une image est $\Theta(nm)$ où n est sa largeur et m sa hauteur.

Voici une fonction qui calcule l'**image miroir** d'une image `im` :

```
let miroir im =
    let contenus_pixels p =
        let (x, y) = p in
        im.contenus_pixels (im.largeur - x - 1, y)
    in
    {im with contenus_pixels = contenus_pixels}
```

Voici des fonctions pour **superposer** deux images `im1` et `im2` de mêmes dimensions en considérant les moyennes de composantes de couleurs entre chaque pixel en mêmes positions :

```
let moyenne_couleurs c1 c2 =
    let moy x y = (x + y) / 2 in
    {rouge = moy c1.rouge c2.rouge; vert = moy c1.vert c2.vert; bleu = moy c1.bleu c2.bleu}

let superposer im1 im2 =
    let contenus_pixels p =
        moyenne_couleurs (im1.contenus_pixels p) (im2.contenus_pixels p)
    in
    {im1 with contenus_pixels = contenus_pixels}
```

Les mêmes remarques précédentes pour les complexités de ces deux fonctions s'appliquent.

Exemple

Voici un exemple d'image qu'il est possible de construire en utilisant les transformations précédentes :

```
let im_alien =
    let noir = {rouge = 0; vert = 0; bleu = 0} in
    let blanc = {rouge = 255; vert = 255; bleu = 255} in
    let cercle x0 y0 r p =
        let (x, y) = p in
        let dx = x - x0 and dy = y - y0 in
        if dx * dx + dy * dy <= r * r then noir else blanc
    in
    let im_oeil_gauche =
        {contenus_pixels = cercle 90 130 20; largeur = 240; hauteur = 240}
        |> complementer
    in
    let im_tete = {contenus_pixels = cercle 120 120 60; largeur = 240; hauteur = 240} in
    let im_yeux = superposer im_oeil_gauche (miroir im_oeil_gauche) in
    superposer im_yeux im_tete
```

3.8.4. Séries génératrices

Nous souhaitons représenter des *séries génératrices*. Ce sont des polynômes en une variable z de degré possiblement infini et à coefficients entiers. En d'autres termes, ce sont des sommes infinies

$$\sum_{n \geqslant 0} \alpha_n z^n = \alpha_0 + \alpha_1 z + \alpha_2 z^2 + \alpha_3 z^3 + \cdots$$

où les α_i sont, dans le cas qui nous intéresse ici, des entiers.

Les séries génératrices sont des outils très importants en informatique. Elles permettent de **coder** de manière compacte des **suites infinies d'entiers**

$$(\alpha_0, \alpha_1, \alpha_2, \alpha_3, \ldots).$$

Exemple

La série génératrice de la suite $(1, 2, 4, 8, 16, \ldots)$ des puissances de 2 est

$$\sum_{n \geqslant 0} 2^n z^n = 1 + 2z + 4z^2 + 8z^3 + 16z^4 + \cdots .$$

Question : comment représenter des séries génératrices ?

Il est impossible de les représenter par des listes à cause du caractère **infini** de ces objets.

Réponse : par une **fonction** qui à tout entier positif n associe le coefficient α_n de z^n.

Ceci est réalisé par le type

```
type series_generatrices = int -> int
```

En effet, pour connaître une série génératrice, il faut et il suffit de connaître pour chaque n le coefficient α_n du monôme $\alpha_n z^n$.

Voici une fonction renvoyant la liste des `n` premiers coefficients d'une série génératrice `sg` :

```
let premiers_coefficients n sg =
    List.init n sg
```

Exemple

La série génératrice des puissances de 2 est ainsi codée par

```
# let puissances_2 =
    fun n -> 1 lsl n;;
val puissances_2 : int -> int = <fun>
```

Nous avons

```
# puissances_2 |> premiers_coefficients 10;;
- : int list = [1; 2; 4; 8; 16; 32; 64; 128; 256; 512]
```

Il existe plusieurs **opérations** sur les séries génératrices :

1. *addition* :

$$\left(\sum_{n\geqslant 0}\alpha_n z^n\right)+\left(\sum_{m\geqslant 0}\beta_m z^m\right)=\sum_{k\geqslant 0}(\alpha_k+\beta_k)z^k \ ;$$

2. *produit d'Hadamard* :

$$\left(\sum_{n\geqslant 0}\alpha_n z^n\right)\boxtimes\left(\sum_{m\geqslant 0}\beta_m z^m\right)=\sum_{k\geqslant 0}\alpha_k\beta_k z^k \ ;$$

3. *multiplication* :

$$\left(\sum_{n\geqslant 0}\alpha_n z^n\right)\cdot\left(\sum_{m\geqslant 0}\beta_m z^m\right)=\sum_{k\geqslant 0}\sum_{i=0}^{k}\alpha_i\beta_{k-i}z^k.$$

Il est possible de les implanter simplement en utilisant des fonctions d'ordre supérieur.

Voici trois manières équivalentes de définir une fonction `addition` qui renvoie la série génératrice résultat de l'**addition** entre les séries génératrices `s1` et `s2` :

```
let addition s1 s2 =
    fun k -> (s1 k) + (s2 k)
```

```
let addition s1 s2 n =
    (s1 n) + (s2 n)
```

```
let addition s1 s2 =
    let res k =
        (s1 k) + (s2 k)
    in
    res
```

Exemple

```
# addition puissances_2 puissances_2 |> premiers_coefficients 10;;
- : int list = [2; 4; 8; 16; 32; 64; 128; 256; 512; 1024]
```

L'implantation du **produit d'Hadamard** utilise les mêmes idées :

```
let produit_hadamard s1 s2 =
    fun k -> (s1 k) * (s2 k)
```

Exemple

```
# produit_hadamard puissances_2 puissances_2 |> premiers_coefficients 10;;
- : int list = [1; 4; 16; 64; 256; 1024; 4096; 16384; 65536; 262144]
```

L'implantation de la multiplication est un peu plus technique dans sa mise en œuvre : elle utilise une fonction auxiliaire récursive.

```
let multiplication s1 s2 =
    let resultat k =
        let rec aux i =
            if i > k then 0
            else (aux (i + 1)) + (s1 i) * (s2 (k - i))
        in
        aux 0
    in
    resultat
```

Exemples

```
# let sg_un = fun _ -> 1;;
val sg_un : 'a -> int = <fun>
# sg_un |> premiers_coefficients 10;;
- : int list = [1; 1; 1; 1; 1; 1; 1; 1; 1; 1]

# let sg_un_2 = multiplication sg_un sg_un;;
val sg_un_2 : int -> int = <fun>
# sg_un_2 |> premiers_coefficients 10;;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
```

3.9. Polymorphisme

3.9.1. Notions de base

Une entité est dite *polymorphe* si elle ou certains de ses composants sont d'un type non fixé.

1. Une *fonction polymorphe* est une fonction paramétrée par au moins un paramètre dont le type peut être quelconque.

Exemple

```
# let vrai x = true;;
val vrai : 'a -> bool = <fun>
```

2. Un *type polymorphe* est un type paramétré.

Exemple

```
# type 'e liste = Vide | Cellule of 'e * 'e liste;;
type 'a liste = Vide | Cellule of 'a * 'a liste
```

3. Une *valeur polymorphe* est une valeur d'un type paramétré dont au moins un paramètre de type reste non spécialisé.

Exemple

```
# Vide;;
- : 'a liste = Vide
```

En OCaml, le polymorphisme est *paramétrique* : ceci signifie qu'un paramètre de type doit pouvoir être remplacé par **n'importe quel** type (et pas seulement par une sous-collection de types).

Corollaire : il n'est pas possible d'écrire des fonctions dont un paramètre peut être d'un type dans une collection de types donnée.

Ainsi, tout paramètre est soit d'un type `t` bien défini, soit de tous les types possibles `'a` (« *tout ou un* »).

De cette manière, pour **déterminer le type d'un paramètre** `x` d'une fonction correctement typée, le système de typage fonctionne (de manière très simplifiée) ainsi :

1. il recherche les occurrences de `x` dans la fonction pour tenter de déterminer son type en fonction de son utilisation ;
2. si cette étape échoue (ou bien s'il n'y a aucune occurrence de `x`), alors `x` est du type le plus général `'a`.

Plusieurs variables de types `'a`, `'b`, `'c`, *etc.*, peuvent coexister dans l'expression de type d'un objet polymorphe.

Beaucoup de fonctions de la bibliothèque standard sont polymorphes. Parmi elles :

- `(=) : 'a -> 'a -> bool` qui teste l'égalité entre deux valeurs d'un même type.
- `(<>) : 'a -> 'a -> bool` qui teste la différence entre deux valeurs d'un même type.
- `compare : 'a -> 'a -> int` qui est telle que `compare x y` renvoie `-1` (resp. `1`) si `x` est strictement inférieure (resp. supérieure) à `y` et `0` sinon.

  Ceci est une **fonction générique de comparaison** entre deux valeurs d'un même type.
- `fst : 'a * 'b -> 'a` qui renvoie la 1re coordonnée d'un couple dont les coordonnées sont de types possiblement différents.
- `snd : 'a * 'b -> 'b` qui renvoie la 2e coordonnée d'un couple dont les coordonnées sont de types possiblement différents.
- `(@@) : ('a -> 'b) -> 'a -> 'b` qui est telle que `f @@ x` renvoie `f x`.
- `(|>) : 'a -> ('a -> 'b) -> 'b` qui est telle que `x |> f` renvoie `f x`.
- `ignore : 'a -> unit` qui est telle que `ignore x` renvoie `()`.

3.9.2. Monoïdes et exponentiation dichotomique

Lorsque x est un nombre et n est un entier positif, le calcul de x^n se fait récursivement par

$$x^n := \begin{cases} 1 & \text{si } n = 0, \\ x^k \times x^k & \text{si } n = 2k, \\ x^k \times x^k \times x & \text{sinon } (n = 2k+1). \end{cases}$$

Ceci se traduit en la fonction

```
let rec puissance x n =
    if n = 0 then
        1
    else
        let tmp = puissance x (n / 2) in
        if n mod 2 = 0 then
            tmp * tmp
        else
            tmp * tmp * x
```

Il faut bien observer en l. 5 la liaison locale de `tmp` pour faire un seul appel récursif au lieu de deux.

L'opération d'exponentiation peut s'appliquer à d'autres objets que des nombres, pourvu que l'on fournisse une opération $\times$ associative et un élément particulier 1, unité pour l'opération $\times$.

En d'autres termes, ces objets doivent former une structure de monoïde.

Nous souhaitons ainsi écrire une fonction polymorphe `puissance_polymorphe` de type

```
('e -> 'e -> 'e) -> 'e -> 'e -> int -> 'e
```

où

- le 1er paramètre de type `'e -> 'e -> 'e` est une fonction codant $\times$;
- le 2e paramètre de type `'e` est l'unité 1 ;
- le 3e paramètre de type `'e` est l'élément `x` ;
- le 4e paramètre de type `int` est l'entier `n` ;
- le type de retour est `'e`.

La valeur renvoyée est x^n.

Nous obtenons ainsi la **fonction polymorphe**

```
let rec puissance_polymorphe op unite x n =
    if n = 0 then
        unite
    else
        let tmp = puissance_polymorphe op unite x (n / 2) in
        if n mod 2 = 0 then
            op tmp tmp
        else
            op (op tmp tmp) x
```

Exemples

```
# puissance_polymorphe (+) 0 1 6;;
- : int = 6

# puissance_polymorphe ( * ) 1 2 10;;
- : int = 1024
```

```
# puissance_polymorphe (^) "" "abb" 4;;
- : string = "abbabbabbabb"

# puissance_polymorphe (||) false false 293898273;;
- : bool = false
```

Il est toujours préférable de définir des types pour représenter des concepts et objets de manière compacte plutôt que de manipuler des fonctions avec beaucoup de paramètres.

Pour cela, on représente les monoïdes au moyen du type enregistrement

```
type 'a monoide = {
    op : 'a -> 'a -> 'a;
    unite : 'a
}
```

La fonction précédente devient

```
let rec puissance_polymorphe monoide x n =
    if n = 0 then
        monoide.unite
    else
        let tmp = puissance_polymorphe monoide x (n / 2) in
        if n mod 2 = 0 then
            monoide.op tmp tmp
        else
            monoide.op (monoide.op tmp tmp) x
```

3.9.3. Listes génériques

Soit le type des **listes génériques**

```
type 'e liste =
    |Vide
    |Cellule of 'e * 'e liste
```

Nous souhaitons écrire une fonction polymorphe `appartient_liste` de type

```
'e -> 'e liste -> bool
```

telle que `appartient_liste x lst` teste la présence de l'élément `x` dans la liste `lst`.

```
let rec appartient_liste x lst =
    match lst with
        |Vide -> false
        |Cellule (y, _) when y = x -> true
        |Cellule (_, reste) -> appartient_liste x reste
```

Ceci se base sur le fait que le test d'égalité `=` est **polymorphe** : il est de type `'a -> 'a -> 'a`.

De plus, comme ce test est **présent de manière générique**, il n'est donc pas nécessaire d'avoir un paramètre supplémentaire pour `appartient_liste` destiné à recevoir une fonction de test d'égalité.

Nous souhaitons maintenant écrire une fonction polymorphe `maximum_liste` qui renvoie le **plus grand élément d'une liste générique**. Cette fonction est de type

`('e -> 'e -> 'e) -> 'e liste -> 'e option`

où

- le 1er paramètre de type `'e -> 'e -> 'e` contient une fonction qui prend comme arguments deux éléments et renvoie le plus grand. Il code l'**opération « max »** d'une relation d'ordre totale ;
- le 2e paramètre de type `'e liste` contient la liste générique dans laquelle la recherche est effectuée.

La valeur renvoyée est une **valeur optionnelle** de type `'e`. La valeur encapsulée est le plus élément de la liste si la liste est non vide et est `None` sinon.

```
let maximum_liste f_max lst =
    let rec aux lst max_prefixe =
        match lst with
            |Vide -> max_prefixe
            |Cellule (x, reste) -> aux reste (f_max max_prefixe x)
    in
    match lst with
        |Vide -> None
        |Cellule (x, reste) -> Some (aux reste x)
```

3.10. Listes

3.10.1. Principes généraux

Le langage OCaml offre un type paramétré `'a list` et une syntaxe appropriée pour manipuler les listes homogènes, sans avoir à les redéfinir.

Une *liste homogène* est une suite finie d'éléments d'un même type.

Les listes sont notées avec des crochets et des points-virgules. Ainsi,

`[e1; e2; ... ; en]`

est une liste contenant, de gauche à droite, les valeurs des expressions `e1`, `e2`, ..., `en`.

La *liste vide* est notée `[]`.

Le type `'a list` est un **type polymorphe** et `[]` est une **valeur polymorphe**.

Exemples

```
# [2; 4; 5 + 3; 16];;
- : int list = [2; 4; 8; 16]
```

```
# [];;
- : 'a list = []
```

L'*opérateur de construction* `::` est un opérateur infixe d'arité deux.

L'expression `e :: lst` est équivalente à l'appel `List.cons e lst`, où `List.cons` est la fonction de type

`'a -> 'a list -> 'a list`

telle que si `e` est un élément et `lst` est une liste, alors `List.cons e lst` a pour valeur la liste qui contient `e` comme 1er élément et ceux de `lst` ensuite.

Exemples

```
# 2 :: [1; 2; 3];;
- : int list = [2; 1; 2; 3]
```

```
# 1 :: 2 :: 3 :: [];;
- : int list = [1; 2; 3]
```

```
# [1; 2] |> List.cons 0 |> List.cons 3;;
- : int list = [3; 0; 1; 2]
```

L'opérateur `::` est **associatif de droite à gauche**.

Exemple

L'expression

`1 :: 2 :: 3 :: []`

désigne l'expression totalement parenthésée

`(1 :: (2 :: (3 :: [])))`

L'opérateur de construction est également un **opérateur de déconstruction** lorsque nous nous en servons dans un filtrage de motifs.

Exemple

```
let tete lst =
    match lst with
        |[] -> failwith "liste vide"
        |e :: _ -> e
```

```
let tete_opt lst =
    match lst with
        |[] -> None
        |e :: _ -> Some e
```

Ces filtrages déconstruisent la liste en argument pour accéder à son 1er élément.
La fonction `failwith` est de type `string -> 'a`. Elle provoque une erreur volontaire.

Exemple

```
let rec un_sur_deux lst =
    match lst with
        |[] -> []
        |[e] -> [e]
        |e1 :: e2 :: reste -> e1 :: (un_sur_deux reste)
```

Ceci renvoie la liste des éléments pris un sur deux à partir de la liste en argument.

Note : il s'agit de **fonctions polymorphes**.

3.10.2. Opérations sur les listes

La bibliothèque standard d'OCaml (`Stdlib`) propose à travers le module `List` diverses fonctions de manipulation de listes. Parmi elles :

Fonction	Type	Valeur renvoyée
`hd`	`'a list -> 'a`	Tête de la liste
`tl`	`'a list -> 'a list`	Liste privée de sa tête
`nth`	`'a list -> int -> 'a`	L'élément de la liste à l'indice donné
`length`	`'a list -> int`	Longueur de la liste
`mem`	`'a -> 'a list -> bool`	Présence de l'élément dans la liste
`rev`	`'a list -> 'a list`	Liste miroir
`append`	`'a list -> 'a list -> 'a list`	Concaténation des deux listes

Exercice : donner (en examinant leurs implantations) les complexités en temps et en espace de ces fonctions relativement au nombres d'éléments dans les listes impliquées.

La plupart des opérations réalisées en pratique sur les listes rentrent dans l'une des catégories suivantes :

1. **créer** une liste selon une spécification ;
2. **transformer** les éléments d'une liste ;
3. **sélectionner** les éléments d'une liste qui vérifient une propriété ;
4. **tester** si tous les (resp. au moins un) élément.s d'une liste vérifie.nt une propriété ;
5. **combiner** les éléments d'une liste pour calculer une valeur ;
6. **permuter** les éléments d'une liste.

Exemples

1. [Création] construire la liste des caractères qui composent une chaîne de caractères, construire la liste des nombres de Fibonacci inférieurs à 512 ;
2. [transformation] multiplier les éléments d'une liste d'entiers par 3, convertir une liste de caractères en la liste des codes ASCII correspondants ;
3. [sélection] obtenir la sous-liste des nombres pairs d'une liste d'entiers ;
4. [test] tester si toutes les chaînes de caractères contenues dans une liste commencent par le caractère `'a'`, tester la présence de l'élément `7` dans une liste d'entiers ;
5. [combinaison] calculer la somme des éléments d'une liste d'entiers, extraire la plus grande valeur d'une liste d'entiers ;
6. [permutation] tri d'une liste, image miroir d'une liste.

Tout ceci se fait à l'aide de **fonctions d'ordre supérieur**.

Pour **créer une liste**, nous spécifions sa longueur n désirée et ses éléments en fonction de leur position i dans la liste, via une fonction f. La valeur de élément d'indice i est $f(i)$.

Une telle fonction f est ainsi de type `int -> 'a`. La **fonction de création** `initialiser` est de type

```
int -> (int -> 'a) -> 'a list
```

et sa définition est

```
let initialiser n f =
    let rec aux i acc =
        if i < 0 then
            acc
        else
            aux (i - 1) (f i :: acc)
    in
    aux (n - 1) []
```

Exemples

```
# initialiser 8 Fun.id;;
- : int list = [0; 1; 2; 3; 4; 5; 6; 7]
```

```
# initialiser 6 ((<=) 2);;
- : bool list = [false; false; true; true; true; true]
```

Nous avons réimplanté ici la fonction `List.init`.

Pour calculer l'**image miroir** d'une liste, le type des éléments qu'elle contient n'est pas important.

Il est préférable de proposer une solution **récursive terminale** :

```
let miroir lst =
    let rec aux lst acc =
        match lst with
            |[] -> acc
            |e :: reste -> aux reste (e :: acc)
    in
aux lst []
```

Exemples

```
# miroir ['a'; 'b'; 'c'];;
- : char list = ['c'; 'b'; 'a']
```

```
# miroir [1; 1; 2; 2; 2; 1];;
- : int list = [1; 2; 2; 2; 1; 1]
```

Nous avons réimplanté ici la fonction `List.rev`.

Une bonne manière de spécifier la manière de **transformer** les éléments d'une liste consiste à donner une fonction f telle que chaque élément x de la liste est transformé en l'élément $f(x)$.

Une telle fonction f est ainsi de type `'a -> 'b`. La **fonction de transformation** `transformer` est de type

```
('a -> 'b) -> 'a list -> 'b list
```

et sa définition est

```
let rec transformer f lst =
    match lst with
        |[] -> []
        |e :: reste -> (f e) :: (transformer f reste)
```

Exemples

```
# transformer (fun x -> x * 3) [1; 2; 3; 4];;
- : int list = [3; 6; 9; 12]
```

```
# transformer int_of_char ['a'; 'b'; 'c'; '1'];;
- : int list = [97; 98; 99; 49]
```

Nous avons réimplanté ici la fonction `List.map`.

Une bonne manière d'**extraire une sous-liste** d'une liste consiste à donner une fonction f qui est telle que $f(x)$ est vrai si l'élément x est à conserver. La fonction f est dans ce cas un *prédicat*.

Une telle fonction f est ainsi de type `'a -> bool`. La **fonction de sélection** `selectionner` est de type

```
('a -> bool) -> 'a list -> 'a list
```

et sa définition est

```
let rec selectionner f lst =
    match lst with
        |[] -> []
        |e :: reste ->
            let suite = selectionner f reste in
            if f e then e :: suite else suite
```

Exemple

```
# selectionner (fun x -> x mod 2 = 0) [13; 8; 9; 8; 6; 15; 2];;
- : int list = [8; 8; 6; 2]
```

Nous avons réimplanté ici la fonction `List.filter`.

Une bonne manière de **tester si tous les éléments** d'une liste vérifient une propriété donnée consiste à représenter cette propriété par un prédicat f qui est tel que $f(x)$ est vrai si x vérifie la propriété.

La **fonction de test universel** `tester_univ` est de type

```
('a -> bool) -> 'a list -> bool
```

et sa définition est

```
let rec tester_univ f lst =
    match lst with
        |[] -> true
        |x :: _ when not (f x) -> false
        |_ :: reste -> tester_univ f reste
```

Exemple

```
# tester_univ (fun u -> u.[0] = 'a') ["a"; "aba"; "abacaba"; "abacabadabacaba"];;
- : bool = true
```

Nous avons réimplanté ici la fonction `List.for_all`.

Une bonne manière de **tester s'il existe un élément** d'une liste qui vérifie une propriété donnée consiste à représenter cette propriété par un prédicat f qui est tel que $f(x)$ est vrai si x vérifie la propriété.

La **fonction de test existentiel** `tester_exist` est de type

```
('a -> bool) -> 'a list -> bool
```

et sa définition est

```
let rec tester_exist f lst =
    match lst with
        |[] -> false
        |x :: _ when f x -> true
        |_ :: reste -> tester_exist f reste
```

Exemple

```
# tester_exist ((=) 7) [0; 1; 1; 2; 3; 5; 8];;
- : bool = false
```

Nous avons réimplanté ici la fonction `List.exists`.

La fonction `List.mem` peut s'implanter par

```
let mem x lst = lst |> tester_exist ((=) x)
```

La problématique ici est la suivante. Nous disposons

1. d'un élément e ;
2. d'une suite d'éléments $(e_1, e_2, \ldots, e_n)$
3. d'une opération binaire $\bullet$;

et nous souhaitons calculer

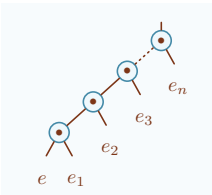
$$(\cdots((e \bullet e_1) \bullet e_2) \bullet \cdots) \bullet e_n.$$

Exemples

Ce problème admet de nombreuses instances, à première vue de nature différente :

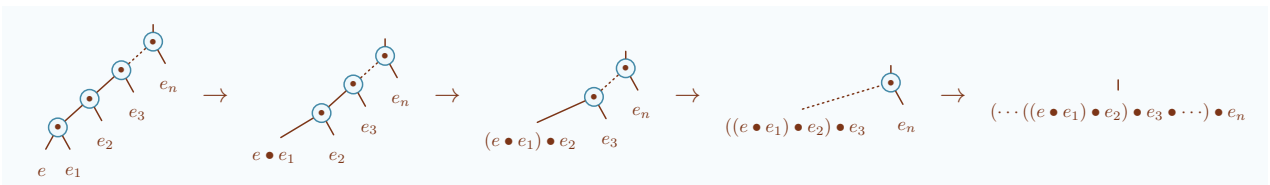
- lorsque $e = 0$, $(e_1, e_2, \ldots, e_n)$ est une suite d'entiers et $\bullet$ est l'opération d'addition des entiers, ceci calcule la somme des éléments de la suite ;
- lorsque $e = \epsilon$ (mot vide), $(e_1, e_2, \ldots, e_n)$ est une suite de chaînes de caractères et $\bullet$ est l'opération de concaténation, ceci calcule la concaténation générale des chaînes de caractères de la suite ;
- lorsque $e = e_1$, $(e_1, e_2, \ldots, e_n)$ est une suite de nombres telle que $n \geqslant 1$ et $\bullet$ est l'opération « $\max$ », ceci calcule la plus grande valeur de la suite.

En d'autres termes, étant donnés l'élément e, la suite d'éléments $(e_1, e_2, \ldots, e_n)$ et l'opération binaire $\bullet$, nous souhaitons **évaluer** l'arbre syntaxique



L'élément e est l'élément initial pour le calcul. Il s'agit de la *graine*.

Ce calcul s'exprime **récursivement** en notant que la valeur recherchée est e lorsque $n = 0$ et sinon, elle est celle calculée dans le cas où la graine est $e \bullet e_1$ et la suite est $(e_2, \ldots, e_n)$:



Ceci s'appelle le *pliage à gauche*.

La suite d'éléments $(e_1, \ldots, e_n)$ est représentée par une liste `lst`, l'opération $\bullet$, par une fonction `f` de type `'a -> 'a -> 'a` et la graine `e`, par un élément de type `'a`.

Ainsi, l'opération de pliage à gauche est de type

```
('a -> 'a -> 'a) -> 'a -> 'a list -> 'a
```

et sa définition est

```
let rec pliage_gauche f e lst =
    match lst with
        |[] -> e
        |x :: reste -> pliage_gauche f (f e x) reste
```

Exemples

```
# pliage_gauche (+) 0 [0; 2; 1; 5; 2; -2; 3];;
- : int = 11
```

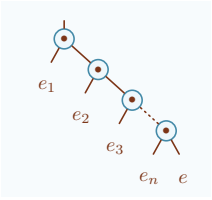
```
# pliage_gauche (^) "" ["mi"; "la"; "re"; "sol"; "do"; "fa"];;
- : string = "milaresoldofa"
```

```
# let lst = [0; 2; 1; 5; 2; -2] in
pliage_gauche max (List.hd lst) (List.tl lst);;
- : int = 5
```

Il est possible de faire essentiellement la même chose, mais pour calculer plutôt

$$e_1 \bullet (e_2 \bullet (e_3 \bullet (\cdots (e_n \bullet e) \cdots)))$$

Ceci se fait en évaluant récursivement l'arbre syntaxique



Il s'agit de l'opération de *pliage à droite* qui est de type

```
('a -> 'a -> 'a) -> 'a list -> 'a -> 'a
```

et dont la définition est

```
let rec pliage_droite f lst e =
    match lst with
        |[] -> e
        |x :: reste -> f x (pliage_droite f reste e)
```

Ces deux opérations de pliage existent sous les noms respectifs de `List.fold_left` et `List.fold_right`.

Les véritables types (qui nous avons précédemment simplifiés dans un but pédagogique) de ces fonctions sont

```
('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

et

```
('b -> 'a -> 'a) -> 'b list -> 'a -> 'a
```

D'un point de vue sémantique, lorsque l'opération $\bullet$ est **associative** et que la graine e **commute** avec tous les éléments, les deux pliages donnent le même résultat.

Il y a une différence d'efficacité : le pliage à gauche est **récursif terminal** alors que le pliage à droite ne l'est pas. En effet, dans le pliage à gauche, **la graine est l'accumulateur**.

Cela dit, le pliage à droite reste intéressant (voire indispensable) lorsque les opérations doivent être réalisées dans un ordre précis. Ceci est illustré dans les prochains exemples.

Les pliages sont des opérations très puissantes : ils permettent d'**émuler** beaucoup de fonctions d'ordre supérieur sur les listes :

Émulation de `List.rev` :

```
let rev lst =
    lst |> List.fold_left (fun res x -> x :: res) []
(* Plus esthétique :      (Fun.flip List.cons) *)
```

Émulation de `List.map` :

```
let map f lst =
    List.fold_right (fun x res -> f x :: res) lst []
```

Émulation de `List.filter` :

```
let filter f lst =
    List.fold_right (fun x res -> if f x then x :: res else res) lst []
```

Émulation de `List.for_all` :

```
let for_all f lst =
    lst |> List.fold_left (fun res x -> f x && res) true
```

Émulation de `List.exists` :

```
let exists f lst =
    lst |> List.fold_left (fun res x -> f x || res) false
```

Les deux utilisations de `List.fold_right` sont importantes ici pour construire une liste dont les éléments apparaissent dans le bon ordre.

En résumé, nous avons étudié et réimplanté les fonctions suivantes du module `List` :

Fonction	Type
`init`	`int -> (int -> 'a) -> 'a list`
`map`	`('a -> 'b) -> 'a list -> 'b list`
`filter`	`('a -> bool) -> 'a list -> 'a list`
`for_all`	`('a -> bool) -> 'a list -> bool`
`exists`	`('a -> bool) -> 'a list -> bool`
`fold_left`	`('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`
`fold_right`	`('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`

Il est important en pratique de veiller à ne pas réimplanter (des variantes de) ces fonctions.

Il existe aussi la fonction `sort` de type

`('a -> 'a -> int) -> 'a list -> 'a list`

renvoyant une version triée d'une liste au moyen d'une fonction de comparaison (1er paramètre).

3.10.3. Exemples de manipulation de listes

Les fonctions opérant sur les listes qui acceptent une **liste en dernier argument** sont adaptées pour être utilisées conjointement à l'**opérateur d'application inversée** `|>`.

Exemples

- Si `lst` est une liste de couples,

```
lst |> List.map (fun (x, y) -> (y, x))
```

  est la liste obtenue en transposant les couples de `lst`.

- Si `lst` est une liste d'entiers,

```
lst |> List.map (fun x -> x * (-2)) |> List.fold_left (+) 0
```

  est la somme des entiers de `lst` multipliés au préalable par `-2`.

- Si `lst` est une liste de listes,

```
lst |> List.map List.length |> List.exists (fun x -> x mod 2 = 0)
```

  teste s'il existe une liste élément de `lst` qui est de longueur paire.

- Si `lst` est une liste de chaînes de caractères,

```
lst |> List.filter (fun u -> u >= "ab") |> List.map String.lowercase_ascii |> List.fold_left (^) ""
```

  est la concaténation des éléments de `lst` supérieurs à `"ab"` puis convertis en minuscules.

Par manipulation de listes, il est possible de proposer la version suivante du calcul de la factorielle d'un entier positif :

```
let fact n =
    List.init n succ |> List.fold_left ( * ) 1
```

```
# List.init 8 fact;;
- : int list = [1; 1; 2; 6; 24; 120; 720; 5040]
```

Lors d'un appel `fact n`, elle procède ainsi :

1. elle commence par construire la liste `[1; 2; ...; n]` via l'appel à `List.init`.

   Notons l'utilisation de la fonction `succ` qui fait qu'en position `i`, la liste construite contient la valeur `i + 1` ;

2. ensuite, elle plie cette liste par multiplication via l'appel à `List.fold_left`.

En suivant des idées similaires à celles de l'exemple précédent, il est possible de proposer la version suivante du calcul du **nombre de Fibonacci** d'un entier positif :

```
let fibonacci n =
    let next (a, b) =
        (b, a + b)
    in
    List.init n (Fun.const ())
    |> List.fold_left (fun res _ -> next res) (0, 1)
    |> fst
```

```
# List.init 12 fibonacci;;
- : int list = [0; 1; 1; 2; 3; 5; 8; 13; 21; 34; 55; 89]
```

Lors d'un appel `fibonacci n`, elle procède ainsi :

1. elle commence par construire la liste support `[(); ...; ()]` de longueur `n` via l'appel à `List.init` ;
2. ensuite, elle plie cette liste en construisant un couple, initialement `(0, 1)`, en le mettant à jour via la fonction `next`.

   Cette fonction `next` contient le mécanisme de calcul de Fibonacci : elle transforme un couple (a_{n-1}, a_n) en le couple $(a_n, a_n + a_{n-1}) = (a_n, a_{n+1})$, où a_m est le m^e nombre de Fibonacci.

3.11. Stratégies d'évaluation

3.11.1. Principes de base

Question : est-ce que l'exécution du programme (en pseudo-code) suivant se termine ?

```
(Fonction intermédiaire.)
Fonction rec f(x) :
    Si x = 0 :
        0
    Sinon :
        x + f(x - 1)
    Fin
Fin
```

```
(Fonction intermédiaire.)
Fonction g(x, y) :
    Si y est pair :
        y
    Sinon :
        x
    Fin
Fin
```

```
(Point d'entrée de l'exécution.)
Début :
    g(f(-1), 0)
Fin
```

Réponse : tout dépend de la *stratégie d'évaluation* du langage, c'est-à-dire, de comment sont évaluées les applications de fonctions à des arguments.

Ici, cela dépend de comment est évaluée l'expression `g(f(-1), 0)` :

1. si l'évaluation de cet appel à `g` a pour prérequis de connaître les valeurs de ses arguments, alors `f` est appliquée à `-1`, ce qui provoque une **non-terminaison** ;
2. sinon, l'expression `f(-1)` n'est pas évaluée car le second argument, `0`, de l'appel à `g` est pair. L'exécution se **termine** dans ce cas.

3.11.2. Appel par valeur

La stratégie d'évaluation en *appel par valeur* consiste, lors de l'application d'une fonction `f` à des expressions `a1`, ..., `an`, à évaluer d'abord `a1`, ..., `an` jusqu'à **obtenir des valeurs**, puis à appliquer `f` sur ces valeurs.

Exemple

Si l'on a une fonction

```
let f x y z =
    x + z
```

l'expression

```
f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4)
```

s'évalue au moyen des étapes suivantes :

```
f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4)  →  f 1 (f 2 1 4) 12  →  f 1 6 12  →  13
```

Par définition, en appel par valeur, tous les arguments d'une fonction lors d'un appel sont évalués.

Ceci entraîne deux inconvénients majeurs :

1. il se peut que certains des arguments de l'appel n'interviennent pas dans la valeur renvoyée par la fonction. Leur **évaluation**, qui a été ainsi faite, a donc été **inutile** (voir l'exemple précédent) ;
2. cette stratégie peut influencer négativement la **terminaison** de certains programmes (voir l'exemple introductif).

Beaucoup de langages utilisent cette stratégie, dont l'OCaml.

3.11.3. Appel par nom

La stratégie d'évaluation en *appel par nom* consiste, lors de l'application d'une fonction `f` à des expressions `a1`, ..., `an`, à **substituer** les `ai` **sans les évaluer** aux occurrences des paramètres de `f` correspondants.

Exemple

Si l'on a une fonction

```
let f x y z =
    x + z
```

l'expression

```
f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4)
```

s'évalue au moyen des étapes suivantes :

```
f (1 * 1) (f (if 1 = 1 then 2 else 3) 1 4) (3 * 4)  →  (1 * 1) + (3 * 4)  →  13
```

Par définition, en appel par nom, tous les arguments d'une fonction sont recopiés tels quels dans son corps lors de son appel.

Ceci peut provoquer une **duplication des calculs**.

Exemple

En effet, si l'on a une fonction

```
let f x y =
    x * x + y
```

l'expression

```
f (4 * 3) (2 * 1)
```

s'évalue en appel par nom au moyen des étapes suivantes :

`f (4 * 3) (2 * 1)` → `(4 * 3) * (4 * 3) + (2 * 1)` → `146`

L'argument `4 * 3` est **évalué** ainsi **deux fois** (au lieu d'une seule que ferait un appel par valeur).

3.11.4. Appel par nécessité

La stratégie en *appel par nécessité* est une **version mémoïsée de l'appel par nom**.

Une fonction est *mémoïsée* si, à chaque premier appel pour un jeu d'arguments donnés, la **valeur qu'elle renvoie** est **enregistrée** dans une table associant les arguments au résultat. Ainsi, tout second appel à la fonction avec le même jeu d'arguments ne provoque pas de réévaluation.

Lors de l'application d'une fonction `f` à des expressions `a1`, ..., `an`, chaque `ai` n'est évalué que si sa valeur est requise pour l'évaluation de l'appel de fonction.

De plus, l'évaluation d'un `ai`, si elle a lieu, est enregistrée. Ainsi, toute occurrence d'un paramètre de `f` correspondant à un `ai` ne redemande pas d'être réévaluée.

De cette manière, l'appel par nécessité prend tous les avantages des appels par valeur et par nom mais aucun de leurs inconvénients.

Néanmoins, cette stratégie n'est applicable que lorsque le **principe de transparence référentielle** est rigoureusement respecté. Elle n'est donc utilisée que par les langages fonctionnels purs.

Rappel : le principe de transparence référentielle est respecté si dans tout programme, il est possible de remplacer une expression par sa valeur sans que cela ne change le résultat construit par l'exécution du programme.

Le langage HASKELL utilise cette stratégie.

3.12. Non mutabilité

3.12.1. Principes généraux

Principalement en programmation fonctionnelle (mais pas uniquement), nous manipulons des données *non mutables* : une fois une donnée construite, il est **impossible de la modifier**.

Corollaire à cela : étant donné une entité `x`, pour obtenir une entité `x'` calculée à partir de `x`, il faut **reconstruire** `x'`.

La non-mutabilité des données présente beaucoup d'avantages :

1. écriture de programmes davantage facilitée et sécurisée ;
2. démonstration de correction de programmes facilitée ;
3. gain de place mémoire.

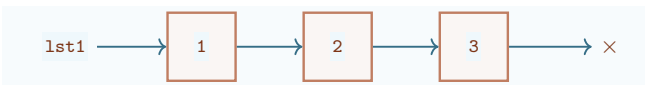
Ces trois avantages s'appuient sur le fait que plusieurs grosses données peuvent **partager** des sous-données en commun, **sans aucune interférence**.

Les listes sont des données non mutables. Toute « modification » d'une liste ne peut se faire que par la **construction d'une liste résultat**.

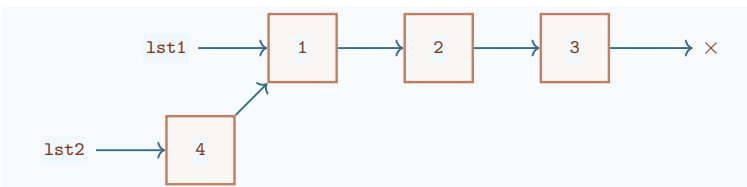
Considérons les phrases

```
# let lst1 = [1; 2; 3];;
# let lst2 = 4 :: lst1;;
```

La 1re créé une liste (trois cellules) en mémoire :



La 2e ne créé qu'une seule cellule et **partage** les trois précédentes :



Considérons la fonction

```
let rec concatener lst1 lst2 =
    match lst1 with
        |[] -> lst2
        |e :: reste -> e :: (concatener reste lst2)
```

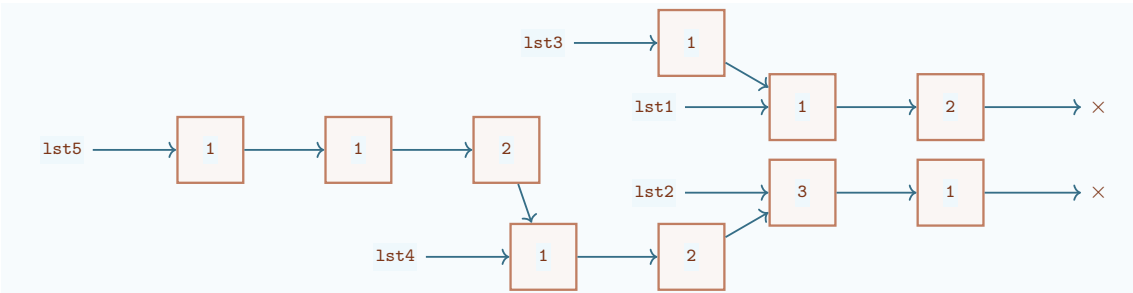
Elle permet de concaténer deux listes (il s'agit d'une implantation de la fonction `List.append`).

Considérons les phrases

```
# let lst1 = [1; 2];;
# let lst2 = [3; 1];;
```

```
# let lst3 = 1 :: lst1;;
# let lst4 = concatener lst1 lst2;;
# let lst5 = concatener lst3 lst4;;
```

Leur exécution produit en mémoire la configuration de partage suivante :



3.12.2. Implantation des files

Nous souhaitons implanter les *files* (piles First In, First Out) dont les éléments sont d'un type quelconque.

On doit pour cela

1. définir un type à un paramètre `file` pour **représenter les files polymorphes** ;
2. définir une constante `vide : 'a file` égale à la **file vide** ;
3. définir une fonction `ajouter : 'a -> 'a file -> 'a file` qui renvoie une nouvelle file prenant en compte de l'**ajout d'un élément** ;
4. définir une fonction `ancien : 'a file -> 'a option` qui renvoie une valeur optionnelle sur **le plus ancien élément** de la file ;
5. définir une fonction `supprimer : 'a file -> 'a file` qui renvoie la file obtenue en **supprimant son plus ancien élément**.

La fonction `ancien` appliquée à la file vide renvoie `None`.

La fonction `supprimer` appliquée à la file vide renvoie la file vide.

L'implantation directe consiste à représenter une file par une liste. Les éléments sont rangés du plus récent au plus ancien.

```
type 'a file = 'a list

let vide = []

let ajouter x f =
    x :: f
```

```
let rec ancien f =
    match f with
        |[] -> None
        |[e] -> Some e
        |_ :: reste -> ancien reste
```

```
let rec supprimer f =
    match f with
        |[] |[_] -> []
        |e :: reste -> e :: supprimer reste
```

En notant par n le nombre d'éléments de la file, nous obtenons les complexités

Fonction	Complexité en temps
`ajouter`	$\Theta(1)$
`ancien`	$\Theta(n)$
`supprimer`	$\Theta(n)$

Une variante possible consiste à ranger les éléments du plus ancien au plus récent. Les complexités en temps obtenues sont complémentaires à celles présentées ici.

Il existe une implantation qui respecte aussi **le principe de non-mutabilité** mais qui est beaucoup plus performante.

Elle se base sur l'idée suivante. Une file est représentée par deux listes in et out.

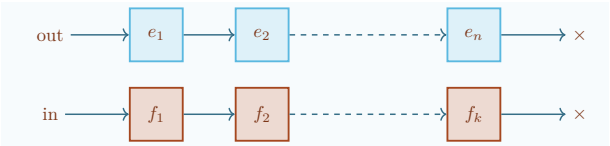
- Les éléments **prêts à sortir** se situent dans out. Ils sont rangés **du plus ancien au plus récent**.
- Les éléments **ajoutés** sont « mis en mémoire tampon » dans in. Ils sont rangés **du plus récent au plus ancien**.

Les opérations sur cette structure de données se décrivent ainsi :

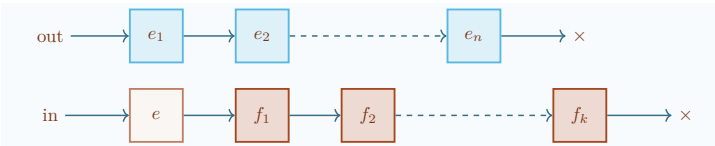
- l'ajout d'un élément e à la file consiste à positionner e en tête de in ;
- la suppression/renvoi du plus ancien élément consiste à supprimer/renvoyer la tête de out si elle est non vide.

  Si out est vide, on remplace out par le **miroir** de in, on vide in et on supprime/renvoie la tête de out.

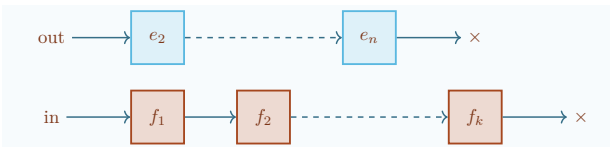
Voici une file dans une situation générique :



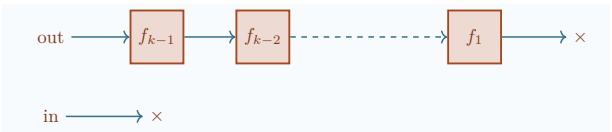
Après l'**ajout** d'un élément e depuis la situation générique, elle devient



Après une **suppression** depuis la situation générique, elle devient



et renvoie e_1 si out n'est pas vide,



et renvoie f_k si out est vide.

Exemple

Opération	in	out	Éléments (plus ancien en 1er)
`vide`	`[]`	`[]`	ϵ
`ajout(1)`	`[1]`	`[]`	1
`ajout(1)`	`[1; 1]`	`[]`	$1, 1$
`ajout(2)`	`[2; 1; 1]`	`[]`	$1, 1, 2$
`ajout(3)`	`[3; 2; 1; 1]`	`[]`	$1, 1, 2, 3$
`supprimer()`	`[]`	`[1; 2; 3]`	$1, 2, 3$
`ajout(4)`	`[4]`	`[1; 2; 3]`	$1, 2, 3, 4$
`ajout(5)`	`[5; 4]`	`[1; 2; 3]`	$1, 2, 3, 4, 5$
`supprimer()`	`[5; 4]`	`[2; 3]`	$2, 3, 4, 5$
`supprimer()`	`[5; 4]`	`[3]`	$3, 4, 5$
`ajouter(6)`	`[6; 5; 4]`	`[3]`	$3, 4, 5, 6$
`supprimer()`	`[6; 5; 4]`	`[]`	$4, 5, 6$
`supprimer()`	`[]`	`[5; 6]`	$5, 6$

```
type 'a file =
    {entree : 'a list ; sortie : 'a list}
```

```
let vide =
    {entree = [] ; sortie = []}
```

```
let ajouter x f =
    {f with entree = x :: f.entree}
```

```
let ancien f =
    match f.sortie with
        |[] -> begin
            match List.rev f.entree with
                |[] -> None
                |e :: _ -> Some e
        end
        |e :: _ -> Some e
```

```
let supprimer f =
    match f.sortie with
        |[] -> begin
            match List.rev f.entree with
                |[] -> f
                |_ :: reste -> {entree = [] ; sortie = reste}
        end
        |_ :: reste -> {f with sortie = reste}
```

Cette implantation est plus efficace que la précédente.

Elle est même strictement plus efficace : les trois opérations ont une complexité en temps en $\Theta(1)$ sauf lorsque out est vide, ce qui demande pour `ancien` et `supprimer` une mise à jour en $\Theta(n)$.

Les fonctions précédentes de manipulation de files s'utilisent aisément en pratique à l'aide de

- l'opérateur d'application d'inversée ;
- la fonction `Option.map : ('a -> 'b) -> 'a option -> 'b option` ;
- la fonction `Option.bind : 'a option -> ('a -> 'b option) -> 'b option`.

Exemple

```
# let f = vide |> ajouter 1 |> ajouter 1 |> ajouter 2 |> ajouter 3;;
val f : int file = {entree = [3; 2; 1; 1]; sortie = []}

# let f = f |> supprimer;;
val f : int file = {entree = []; sortie = [1; 2; 3]}

# let f = f |> ajouter 4 |> ajouter 5;;
val f : int file = {entree = [5; 4]; sortie = [1; 2; 3]}

# f |> ancien |> Option.map (fun x -> Printf.printf "Valeur : %d\n" x);;
Valeur : 1
- : unit option = Some ()
```

3.13. Preuves

3.13.1. Validité d'un programme

Un **programme** p est *valide* si ce qu'il calcule est conforme à ce qui est attendu.

Ce qui est attendu est décrit par une *spécification*.

Exemples

Voici quelques spécifications de programmes :

- le programme p_1 affiche tous les diviseurs de 60 ;
- le programme p_2, sur l'argument entier n, affiche « oui » si n est un nombre premier et « non » sinon.

Cette notion s'étend aux **fonctions** : une fonction f est *valide* si elle est conforme à une *spécification*. Elle décrit ce que f renvoie lorsque appliquée à des arguments a_1, ..., a_n.

Exemples

Voici quelques spécifications de fonctions :

- la fonction f_1 renvoie la somme des éléments de la liste d'entiers sur laquelle elle est appliquée ;
- la fonction f_2 teste si l'arbre binaire sur lequel elle est appliqué est équilibré.

Un *test* (*unitaire*) d'une fonction f est un couple (e, r) où e est une expression dans laquelle f intervient, pour laquelle le résultat r conforme à la spécification de f est connu.

Il consiste à comparer e avec la valeur attendue r. Ce test est *positif* lorsque la valeur de e est bien r et est *négatif sinon*.

Exemple

Soit la fonction

```
let est_premier n =
    List.init (n - 2) ((+) 2)
    |> List.exists (fun k -> n mod k = 0)
    |> not
```

spécifiée par « `est_premier` renvoie `true` ssi elle est appliquée à un entier $n \geqslant 2$ premier ».

Un test de `est_premier` est composé de l'expression

```
List.init 20 ((+) 2) |> List.filter est_premier
```

et de la valeur attendue

```
[2; 3; 5; 7; 11; 13; 17; 19]
```

Ce test est positif.

Dans le cas où f est une fonction de **domaine fini**, un ensemble de tests parcourant toutes les entrées possibles peut montrer que la fonction est valide. Cet ensemble de tests est alors qualifié de *couvrant*.

Exemple

Soient le type et la fonction

```
type joueurs = Blancs | Noirs

let adversaire j =
    match j with
        |Blancs -> Noirs
        |Noirs -> Blancs
```

spécifiée par « `adversaire` renvoie l'adversaire du joueur sur lequel elle est appliquée ».

Son domaine est fini. Les deux tests

`(adversaire Blancs, Noirs)` et `(adversaire Noirs, Blancs)`

forment un ensemble de tests couvrant.

Dans le cas contraire, un ensemble de tests **ne démontre pas la validité** d'une fonction. Il peut seulement **démontrer son invalidité**.

Malgré cette limitation pour les fonctions à domaines infinis (qui sont les plus courantes), les tests sont utiles pour

- détecter des **erreurs** lors de la **phase initiale** de programmation ;
- détecter des **erreurs émergentes** lors des **mises à jour** du programme ;
- participer à la **documentation** d'une fonction en illustrant une façon de l'utiliser.

Écrire de bons tests demande

- une **compréhension parfaite de la spécification** de la fonction ;
- un moyen de **connaître à l'avance** ce qu'elle doit calculer ;
- une **bonne imagination** pour concevoir des tests significatifs et non redondants.

Il est important que chaque test mette au défi la fonction f testée sur le moins d'aspects possibles à la fois. Ceci est utile car, lorsque le test échoue, il devient facile de délimiter le problème contenu dans l'implantation de f.

Plusieurs méthodes sont possibles pour mettre des tests en place :

- une **approche artisanale** consiste à utiliser une fonction outil de la forme

```
let test nom e r =
    Printf.printf "Test %s : " nom;
    if e = r then
        print_endline "[SUCCES]"
    else
        print_endline "[ECHEC]"
```

  et à disposer dans (par exemple) un fichier `Tests.ml` une fonction principale de test.

Exemple

```
let () =
    test "est_premier 2" (est_premier 2) true;
    test "est_premier 15" (est_premier 15) false;
    test "est_premier 23" (est_premier 23) true
```

- Une approche plus robuste consiste à utiliser des **bibliothèques dédiées** comme `Alcotest`, `MDX` `OUnit` ou `QCheck`. Ceci s'articule bien avec `dune` qui peut accueillir des fichiers de test.

3.13.2. Équivalence de fonctions

Deux fonctions `f1` et `f2` sont *égales*, noté `f1 = f2`, si leur code est **syntaxiquement identique**.

Ce n'est pas une notion très intéressante en pratique.

Deux fonctions `f1` et `f2` sont *équivalentes*, noté `f1 ≡ f2`, si elles sont **extensionnellement équivalentes**. C'est le cas lorsque

- `f1 = f2` si `f1` et `f2` sont des valeurs ;
- pour tout argument `a`, `f1 a ≡ f2 a`, si `f1` et `f2` admettent au moins un argument.

Les fonctions `f1` et `f2` se comportent alors de la même façon : en notant `n` le nombre d'arguments de `f1` et `f2`, pour tous arguments `a1`, ..., `an`, `f1 a1 ...an` et `f2 a1 ...an` renvoient la même valeur.

Exemple

Les fonctions

```
let doubler_1 lst =
    lst |> List.map (fun x -> x * 2)
```

et

```
let rec doubler_2 lst =
    match lst with
        |[] -> []
        |x :: lst' -> x * 2 :: doubler_2 lst'
```

sont équivalentes. Elles sont soumises à **la même spécification** qui est de renvoyer la liste d'entiers en doublant chacun de ses éléments.

Le système de types du langage assure à lui seul une bonne partie de la spécification d'une fonction `f`. Il permet souvent de déterminer des propriétés plus fines que seulement la nature des paramètres et du renvoi de `f`.

Dans certains cas très favorables, le type d'une fonction impose même de manière unique son comportement.

Exemples

- Une fonction `f` de type `'a -> 'a` est forcément équivalente à l'identité.
- Une fonction `f` de type `'a * 'b -> 'a` est forcément équivalente à `fst`.
- Une fonction `f` de type `'a -> ('a -> 'b) -> 'b` est forcément équivalente à la fonction d'application inversée `(|>)`.
- Une fonction `f` de type `'a list -> 'a` renvoie nécessairement l'un des éléments de la liste sur laquelle elle est appliquée.
- Une fonction `f` de type `'a list -> 'a list` renvoie nécessairement une liste constituée d'éléments de la liste sur laquelle elle est appliquée.

En pratique cependant, les types ne suffisent pas pour démontrer qu'une fonction suit une spécification donnée.

Le système de types **polymorphe paramétrique** d'OCaml, bien que puissant et bien meilleur que beaucoup de ses concurrents, est finalement assez limité.

Exemples

En utilisant seulement le système de types, il est impossible d'induite des contraintes sur

- les longueurs des listes (le type des listes de longueurs paires, de longueurs inférieures à n ou de longueur non nulle n'existe pas) ;
- les intervalles auxquels des nombres doivent appartenir ;
- les contraintes structurelles d'une valeur (listes ayant des contraintes locales, comme le fait d'être triées, arbres qui évitent certains motifs).

Il existe des systèmes de types beaucoup plus performants qui le permettent. C'est le cas des **types dépendants**. Dans un tel système de types, il est par exemple possible de spécifier au niveau des types qu'une fonction accepte et renvoie un arbre binaire équilibré.

3.13.3. Démonstrations d'équivalence

Pour **démontrer** qu'une fonction `f` vérifie une spécification, nous démontrons que `f` est équivalente à une fonction g qui est une **fonction hypothétique** vérifiant la spécification.

Commençons par un exemple quasi immédiat.

Exemple

Soit la spécification « la fonction renvoie le dernier élément de la liste non vide ».

Soit la fonction

```
let dernier lst =
    lst |> List.rev |> List.hd
```

Appelons g une fonction hypothétique qui répond à la spécification.

Ainsi, pour toute liste $[e_1, \dots, e_{n-1}, e_n]$ non vide (donc $n \geqslant 1$), nous avons $g\,[e_1, \dots, e_{n-1}, e_n] = e_n$.

D'après la définition de `dernier`, nous avons

$$\texttt{dernier}\,[e_1, \dots, e_{n-1}, e_n] = [e_1, \dots, e_{n-1}, e_n]\,\texttt{|> List.rev |> List.hd}.$$

D'après les spécifications (considérées comme vérifiées) de `List.rev` et `List.hd`, nous avons

$$[e_1, \dots, e_{n-1}, e_n]\,\texttt{|> List.rev |> List.hd} = [e_n, e_{n-1}, \dots, e_1]\,\texttt{|> List.hd} = e_n.$$

Ainsi, les fonctions `dernier` et g sont équivalentes. La fonction `dernier` vérifie donc la spécification annoncée.

Il existe un outil très puissant pour démontrer l'équivalence de fonctions opérant sur des valeurs d'un **type somme** : l'*induction structurelle*. Voici quelques définitions pour l'introduire.

Soit T un **type somme** défini par l'intermédiaire de **constructeurs** C.

Si C est un constructeur du type T,

- l'*arité* de C est le nombre d'arguments de type T qu'il reçoit ;
- si C est un constructeur d'arité n de T et x_1, ..., x_n sont des éléments de type T, alors $C(x_1, \ldots, x_n)$ désigne tout élément de type T obtenu en utilisant le constructeur C avec les arguments x_1, ..., x_n et éventuellement d'autres arguments de types différents de T.

Exemple

Soit le type somme des arbres unaires binaires (sans contrainte supplémentaire ici) :

```
type 'n arbres_unaires_binaires =
    |Feuille of int
    |NoeudUnaire of 'n * ('n arbres_unaires_binaires)
    |NoeudBinaire of 'n * ('n arbres_unaires_binaires) * ('n arbres_unaires_binaires)
```

Les constructeurs `Feuille`, `NoeudUnaire` et `NoeudBinaire` sont d'arités respectives 0, 1 et 2.

Si `t1` et `t2` sont de type `'n arbres_unaires_binaires`, alors `NoeudBinaire(t1, t2)` désigne tout élément de type `'n arbres_unaires_binaires` de la forme `NoeudBinaire (x, t1, t2)` où `x` est de type `'n`.

L'élément `NoeudBinaire (Feuille 1, NoeudUnaire ('x', Feuille 2))` est un exemple d'une telle valeur.

Soit T un type somme et P une propriété sur T.

Un constructeur C d'arité n de T vérifie la *propriété d'induction structurelle* si pour tous éléments x_1, ..., x_n de type T, $P(x_1) \wedge \cdots \wedge P(x_n)$ implique $P(C(x_1, \ldots, x_n))$.

Terminologie :

- lorsque C est d'arité 0, cette propriété est appelée **cas de base** ;
- lorsque C est d'arité $n \geqslant 1$, cette propriété est appellée **cas d'hérédité**.

Le *principe d'induction structurelle* stipule que si tout constructeur de T vérifie la propriété d'induction structurelle, alors $P(x)$ est vrai pour tout élément x de type T.

En définissant adéquatement P relativement à la spécification f d'une fonction, ce principe est un **outil très puissant** pour démontrer que f vérifie cette spécification.

Exemple

Soit le type

```
type arbres =
    |Feuille
    |Noeud of arbres * arbres
```

En notant $\mathrm{n}(\texttt{t})$ (resp. $\mathrm{f}(\texttt{t})$) le nombre de nœuds (resp. feuilles) de `t`, soit la propriété P telle que $P(\texttt{t})$ est vrai si $\mathrm{f}(\texttt{t}) = \mathrm{n}(\texttt{t}) + 1$.

Démontrons $P(\texttt{t})$ pour tout `t` de type `arbres` en utilisant le principe d'induction structurelle.

- (Constructeur `Feuille`.) Nous avons bien $\mathrm{f}(\texttt{Feuille}) = \mathrm{n}(\texttt{Feuille}) + 1$ car $\mathrm{f}(\texttt{Feuille}) = 1$ et $\mathrm{n}(\texttt{Feuille}) = 0$. Ainsi, nous avons bien $P(\texttt{Feuille})$.
- (Constructeur `Noeud`.) Soient `t1` et `t2` deux éléments de type `arbres` tels que $P(\texttt{t1})$ et $P(\texttt{t2})$. Soit $\texttt{t} = \texttt{Noeud (t1, t2)}$. Comme les feuilles d'un arbre binaire sont exclusivement situées dans ses deux sous-arbres, $\mathrm{f}(\texttt{t}) = \mathrm{f}(\texttt{t1}) + \mathrm{f}(\texttt{t2})$. De plus, comme $P(\texttt{t1})$ et $P(\texttt{t2})$,

$$\mathrm{f}(\texttt{t1}) + \mathrm{f}(\texttt{t2}) = \big(\mathrm{n}(\texttt{t1}) + 1\big) + \big(\mathrm{n}(\texttt{t2}) + 1\big) = \mathrm{n}(\texttt{t1}) + \mathrm{n}(\texttt{t2}) + 2$$

  Comme les nœuds d'un arbre binaire sont exclusivement situés dans ses deux sous-arbres ou sa racine, $\mathrm{n}(\texttt{t}) = \mathrm{n}(\texttt{t1}) + \mathrm{n}(\texttt{t2}) + 1$. Ceci montre $\mathrm{f}(\texttt{t}) = \mathrm{n}(\texttt{t}) + 1$ et donc $P(\texttt{t})$.

Une `liste` est un type somme où `[]` est un constructeur d'arité 0 et `::` est un constructeur d'arité 1.

Exemple

Soit la fonction

```
let rec dernier lst =
    match lst with
        |[] -> None
        |[x] -> Some x
        |_ :: lst' -> dernier lst'
```

Soit g la fonction telle que g `lst` renvoie `None` si `lst` est vide et `Some x` sinon, où `x` est le dernier élément de `lst`.

Soit la propriété P telle que $P($`lst`$)$ si `dernier lst` renvoie la même résultat que g `lst`.

Démontrons $P($`lst`$)$ pour toute liste `lst` en utilisant le principe d'induction structurelle.

- (Constructeur `[]`.) L'appel `dernier []` s'évalue en `None`. Ceci montre $P($`[]`$)$.
- (Constructeur `::`.) Soit `lst'` une liste telle que $P\big($`lst'`$\big)$.
  Si `lst'` $=$ `[]`, l'appel `dernier lst` s'évalue en `Some x` où `x` est la tête de `lst`. Comme dans ce cas, `x` est aussi le dernier élément de `lst`, nous avons $P($`lst`$)$.
  Sinon, l'appel `dernier lst` s'évalue en `dernier lst'`. Comme $P\big($`lst'`$\big)$, l'expression `dernier lst'` s'évalue en `Some x'` où `x'` est le dernier élément de `lst'`. Comme `lst'` est la queue de `lst`, ces deux listes ont le même dernier élément. Ainsi, nous avons $P($`lst`$)$.

Considérons le type des *entiers de Peano* défini par

```
type naturels_peano =
    |Zero
    |Succ of naturels_peano
```

Il représente l'ensemble $\mathbb{N}$, où chaque $n \in \mathbb{N}$ est codé par l'élément `Succ (...(Succ (Succ Zero)) ...)` de type `naturels_peano` avec n occurrences de `Succ`. Par exemple, l'entier 2 est codé par `Succ (Succ Zero)`.

Soit P une propriété sur `naturels_peano`. Ainsi, P est une **propriété sur les entiers naturels**.

D'après le principe d'induction structurelle, pour démontrer que $P($`n`$)$ est vrai pour tout élément `n` de type `entiers_peano`, il faut démontrer

- que $P($`Zero`$)$ est vrai (**cas de base**) ;
- que $P($`n`$)$ implique $P($`Succ n`$)$ pour tout élément `n` de type `entiers_peano` (**cas d'hérédité**).

Nous reconnaissons alors le **principe d'induction** habituel.

Il est finalement un **cas particulier** du principe d'induction structurelle.

4. Programmation logique

4.1. Initiation au Prolog

4.1.1. Informations générales

La *programmation logique* est un paradigme de programmation basé sur la **logique des prédicats et la logique du 1er ordre**.

Il n'y a pas de fonction dans ce paradigme. L'objet central est le **prédicat**.

Dans ce paradigme, des **faits** et des **règles** sont déclarés, ce qui forme une base de connaissances. Par la suite, nous pouvons poser des questions au système par l'intermédiaire de **requêtes**.

Voici un historique succinct.

- Années 1960 : premières apparitions du paradigme de programmation logique.
- 1969 : conception du langage PLANNER par C. Hewitt, considéré comme le 1er langage de programmation logique.
- 1972 : conception du langage PROLOG par A. Colmerauer et P. Roussel.
- Fin des années 1970 à aujourd'hui : développement d'extensions du PROLOG (amélioration des performances, intégration d'une approche orientée objets, prise en compte des systèmes distribués, *etc.*).

La programmation logique a beaucoup été utilisée pour la mise au point de **systèmes experts**, de **résolveurs de problèmes** et de **traitement automatique des langues**.

En programmation logique, un **fait** est une vérité déclarée comme telle. Ce sont des vérités relatives au contexte de travail.

Exemples

Voici quelques faits.

- Le nombre 24 est pair.
- Randy est le père de Stan.
- La ligne de métro 1 s'arrête à la station Z.
- La ligne de métro 1 ne s'arrête pas à la station Z.

Certaines vérités sont plus complexes et nécessitent une abstraction par le biais de **variables universelles**. Elles peuvent être **instanciées** dans certains cas.

Exemples

Voici quelques vérités nécessitant des variables.

- La ligne de métro 1 s'arrête à toutes les stations.
  Ceci est équivalent à dire que la ligne de métro 1 s'arrête à la station x.
- Les chercheur.e.s x et y du laboratoire ont publié un article en commun.
  Cela signifie que toute paire de chercheur.e.s du laboratoire a publié un article en commun.

Une **règle** est un fait soumis à des conditions. Il est de la forme

$$\text{si } C \text{ alors } F$$

où C est une condition qui, lorsqu'elle est vérifiée, assure que F est un fait.

Exemples

Voici quelques règles.

- Si l'entier n est divisible par 4, alors n est pair.
- Si la ligne de métro x est la 2 ou la 3, alors x s'arrête à la station Z.
- Si la note d'examen n_1 de l'étudiant.e e est supérieure à 72, alors le cours est validé.
  Si la note d'examen n_1 de l'étudiant.e e est supérieure à 29 et la note de devoir n_2 de l'étudiant e supérieure à 99, alors le cours est validé.

Remarque : un **fait** est finalement une règle **si** C **alors** F dont la **condition** C est **toujours vraie**. Cela revient bien à dire que F est vraie.

Une **requête** est une question posée, relativement à un ensemble de faits et de règles. Le processus qui crée une réponse **affirmative** ou **négative** est sa **résolution**.

La résolution est affirmative lorsque le système parvient à **démontrer** la requête. Il se base sur les faits et les règles qui le constituent.

Lorsque le système ne parvient pas à démontrer que la requête est vraie, la résolution est négative. Il s'agit de la **négation par l'échec**.

Exemples

Soient les faits et règles suivants :

1. Randy est le père de Stan.
2. Randy est le père de Shelly.
3. Sharon est la mère de Stan.
4. Si x est le père de y et x est le père de z, alors y et z sont dans la même fratrie.

Voici quelques requêtes et leurs résolutions.

- Soit la requête « Stan et Shelly sont dans la même fratrie. ». Elle admet une résolution affirmative car en posant $x = \texttt{Randy}$, $y = \texttt{Stan}$ et $z = \texttt{Shelly}$, la règle 4 s'applique. Les règles 1 et 2 montrent que la prémisse de cette règle est vraie, entrainant sa conclusion.
- La requête « Sharon est la mère de Shelly. » ne peut pas être démontrée. Sa résolution est donc négative.

Une requête peut contenir des **variables**. Dans ce cas, le processus de résolution va chercher l'**ensemble de ses solutions**. Il s'agit de l'ensemble des des **spécialisations** des variables qui donnent des requêtes dont la résolution est affirmative.

Exemples

Soient les faits et règles suivants :

1. Randy est le père de Stan.
2. Randy est le père de Shelly.
3. Sharon est la mère de Stan.
4. Si x est le père de y et x est le père de z, alors y et z sont dans la même fratrie.

- La requête « a est le père de b. » admet comme ensemble de solutions

$$\{[a = \texttt{Randy}, b = \texttt{Stan}],\ [a = \texttt{Randy}, b = \texttt{Shelly}]\}.$$

- La requête « a et b sont dans la même fratrie. » admet comme ensemble de solutions

$$\{[a = \texttt{Stan}, b = \texttt{Shelly}],\ [a = \texttt{Shelly}, b = \texttt{Stan}],\ [a = \texttt{Stan}, b = \texttt{Stan}],\ [a = \texttt{Shelly}, b = \texttt{Shelly}]\}.$$

4.1.2. Programmer en Prolog

Nous utilisons l'implantation SWI-Prolog. Il en existe d'autres (qui ne seront pas considérés dans ce cours).

Voici quelques outils et références en lien :

- site du projet : https://www.swi-prolog.org/ ;
- plateforme interactive en ligne : https://swish.swi-prolog.org/ ;
- sur les systèmes Linux, l'exécutable est `swipl` ;
- voir les références sur la programmation logique données dans les 1res pages de ce cours.

Nous utiliserons `swipl` principalement mais la plateforme interactive est très utile. Elle contient en plus cela de nombreux exemples.

Le programme `swipl` travaille sur des fichiers d'extension `.pl`. Il s'agit d'un **interpréteur** qui **résout des requêtes**.

Voici quelques prédicats prédéfinis utiles :

- `consult("A.pl").` importe les déclarations du fichier `A.pl` dans la session ;
- `reconsult("A.pl").` importe à nouveau et sans les dupliquer les déclarations du fichier `A.pl` dans la session ;
- `make.` importe à nouveau tout fichier déjà importé qui a subi une modification depuis sa dernière importation ;
- `listing.` affiche toutes les règles de la session ;
- `trace.` active des informations de débogage. La commande `notrace.` permet de les désactiver ;
- `help(X).` affiche de la documentation sur l'entité `X` ;
- `halt.` quitte la session.

Il est possible de charger directement plusieurs fichiers `F1.pl`, ..., `Fn.pl` par la commande

```
swipl -s F1.pl -s F2.pl ... -s Fn.pl
```

Il est aussi possible d'ajouter des options `-g Q` pour demander la résolution de la requête `Q`.

Un *terme* est défini récursivement comme étant

- soit une *variable*, qui commence par une majuscule ou un tiret bas ;
- soit un *atome*, qui commence par une minuscule, attaché à une suite de termes placés entre parenthèses et séparés par des virgules.

L'*arité* d'un atome `a` est le nombre `n` de termes sur lequel il s'applique. Ceci est noté `a/n`.

Exemples

- L'assemblage syntaxique

  ```
  aime(Chat, croquettes(marque(minou_maxou), type(chats_sportifs)))
  ```

  est un terme.

  L'atome `aime` est d'arité 2. Son premier argument est la variable `Chat`.

  L'atome `chats_sportifs` est d'arité 0. Il ne prend aucun argument et les parenthèses sont donc omises.

- L'assemblage syntaxique

  ```
  egaux(moins(X, Y), plus(moins(moins(X)), moins(Y)))
  ```

  possède plusieurs occurrences différentes de l'atome `moins`. Certaines vérifient `moins/2` et d'autres, `moins/1`. Ce sont des atomes différents.

Une *clause* est un assemblage syntaxique de la forme

```
t :- t1, ..., tn.
```

où `t` est un terme appelé *tête* et `t1, ..., tn` est une conjonction de termes appelée *corps*.

Cette clause exprime la règle

si t_1 et ... et t_n alors t.

Pour exprimer un fait, la syntaxe devient simplement

```
t.
```

Exemples

- La clause (qui est un fait)

```
pere(randy, stan).
```

  exprime que Randy est le père de Stan.

- La clause (qui est une règle)

```
fratrie(X, Y) :- pere(Z, X), pere(Z, Y).
```

  exprime le fait que `X` et `Y` sont dans la même fratrie s'il existe un père `Z` commun.

Un *prédicat* est une **suite de clauses** dont la tête est formée du même atome.

Exemple

Les trois clauses

```
pere(randy, stan).
pere(randy, shelly).
pere(marvin, randy).
```

définissent le prédicat `pere`.

Un *programme* est une **suite de prédicats**.

Exemple

Les trois prédicats

```
pere(randy, stan).
pere(randy, shelly).
pere(marvin, randy).

fratrie(X, Y) :- pere(Z, X), pere(Z, Y).

grand_pere(X, Y) :- pere(X, Z), pere(Z, Y).
```

forment un programme.

Une *requête* est un assemblage syntaxique de la forme

```
t1, ..., tn.
```

où `t1, ..., tn` est une conjonction de termes.

Il est possible de la mentionner dans l'interpréteur ou bien via l'option `-g` comme vu précédemment.

La *résolution* d'une requête est le processus qui consiste à **démontrer qu'elle est vraie**. Plusieurs types de réponses sont possibles :

- lorsque la requête est une conséquence des prédicats du programme, c'est-à-dire que chaque terme `t` de la conjonction peut être démontré, `true.` est imprimé ;
- lorsque la requête n'est pas une conséquence des prédicats du programme, `false.` est imprimé ;
- lorsque des variables figurent dans la requête, toutes les façons de rendre la requête conséquence des prédicats du programme en spécialisant les variables sont imprimées.

Lorsqu'une requête admet **plusieurs solutions**, le symbole `;` est imprimé juste après avoir imprimé une solution. L'utilisateur peut presser **ENTRÉE** pour interrompre la recherche ou bien **ESPACE** pour **rechercher la prochaine solution** potentielle.

Si la résolution s'engage dans une voie qui **ne démontre pas** la requête, `false.` est imprimé (même si la résolution de la requête est affirmative via une solution trouvée précédemment).

Exemple

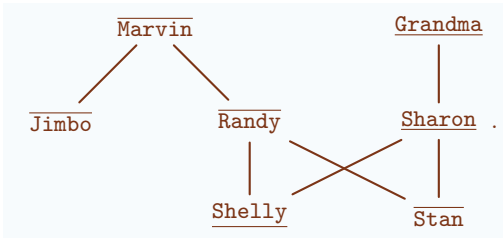
Considérons le programme

```
femelle(grandma).
femelle(sharon).
femelle(shelly).

male(marvin).
male(randy).
male(jimbo).
male(stan).

parent(marvin, randy).
parent(marvin, jimbo).
parent(grandma, sharon).
parent(randy, shelly).
parent(randy, stan).
parent(sharon, shelly).
parent(sharon, stan).
```

Ces faits modélisent l'« arbre » généalogique



Il se lit de haut en bas. Les noms de la forme NOM correspondent à des femelles et les noms de la forme $\overline{\text{NOM}}$, à des mâles.

Dans les exemples qui vont suivre, nous allons considérer des prédicats supplémentaires et observer les réponses obtenues à plusieurs requêtes.

Exemples

Voici quelques requêtes simples et leurs réponses :

- ```
  ?- male(randy). % ENTRÉE
  true.
  ```

  La réponse est `true.` car le système démontre que `male(randy)` est vrai comme conséquence qu'il s'agit d'un fait.

- ```
  ?- parent(marvin, randy). % ENTRÉE
  true.
  ```

  La réponse est `true.` car le système démontre que `parent(marvin, randy)` est vrai comme conséquence qu'il s'agit d'un fait.

- ```
  ?- parent(marvin, stan). % ENTRÉE
  false.
  ```

  La réponse est `false.` car le système ne parvient pas à démontrer que `parent(marvin, stan)` est vrai. La réponse à cette requête est donc négative.

- ```
  ?- male(stan), parent(sharon, shelly). % ENTRÉE
  true.
  ```

  La réponse est `true.` car le système démontre que les deux sous-requêtes sont vraies (il s'agit d'un « et » logique).

Exemples

Voici quelques requêtes avec des variables et leurs réponses :

- ```
  ?- male(X). % ENTRÉE
  X = marvin ; % ESPACE
  X = randy ; % ESPACE
  X = jimbo ; % ESPACE
  X = stan.
  ```

  Cette requête contient une variable `X`. Sa résolution consiste à trouver toutes les spécialisations α de `X` qui rendent `male(`α`)` affirmative.

- ```
  ?- parent(marvin, X). % ENTRÉE
  X = randy ; % ESPACE
  X = jimbo.
  ```

  Le 1er argument de `parent` est un atome et le 2e est une variable. Les spécialisations qui rendent la requête affirmative sont données.

- ```
  ?- parent(marvin, X), parent(X, stan). % ENTRÉE
  X = randy ; % ESPACE
  false.
  ```

  La même variable apparaît dans les deux sous-requêtes. Nous cherchons donc les spécialisations α de `X` telles que Marvin est parent de α et α est parent de Stan.

  Le `false` signifie que la recherche a été poursuivie sans apporter de nouvelle solution.
  ```

Exemples

Voici quelques requêtes avec des variables et leurs réponses :

```
?- parent(marvin, X), parent(Y, stan). % ENTRÉE
X = Y, Y = randy ; % ESPACE
X = randy, Y = sharon ; % ESPACE
X = jimbo, Y = randy ; % ESPACE
X = jimbo, Y = sharon.
```

Cet exemple ressemble au précédent, mais une 2e variable est considérée dans la 2e requête.

Cela donne comme solutions les spécialisations α et β de `X` et `Y` telles que Marvin est parent de α et β est parent de Stan.

```
?- parent(X, Y), parent(Y, X). % ENTRÉE
false.
```

Cette requête est **négative** car le système ne trouve **aucun exemple** de α et β tels que α est parent de β et β est parent de α.

```
?- parent(X, shelly), femelle(X). % ENTRÉE
X = sharon.
```

Nous cherchons les α tels que α est parent de Shelly et est une femelle.

Exemples

Ajoutons au programme précédent les deux prédicats

```
pere(X, Y) :-
    male(X),
    parent(X, Y).
```

Ce prédicat met `X` et `Y` en relation si `male(X)` et `parent(X, Y)`. C'est le cas lorsque `X` est un mâle et `X` est parent de `Y`.

```
mere(X, Y) :-
    femelle(X),
    parent(X, Y).
```

Ce prédicat met `X` et `Y` en relation si `X` est une femelle et `X` est parent de `Y`.

Voici quelques requêtes bâties autour de ces prédicats :

- ```
  ?- pere(X, stan). % ENTRÉE
  X = randy ; % ESPACE
  false.
  ```

  Il n'y a que Randy qui soit le père de Stan.

- ```
  ?- mere(X, Y). % ENTRÉE
  X = grandma, Y = sharon ; % ESPACE
  X = sharon, Y = shelly ; % ESPACE
  X = sharon, Y = stan ; % ESPACE
  false.
  ```

  Ceci donne toutes les spécialisations de `X` et `Y` telles que `X` est mère de `Y`.

Exemples

Ajoutons au programme précédent les trois prédicats

```
grand_parent(X, Y) :-
    parent(X, Z),
    parent(Z, Y).
```

```
grand_pere(X, Y) :-
    male(X),
    grand_parent(X, Y).
```

```
grand_mere(X, Y) :-
    femelle(X),
    grand_parent(X, Y).
```

Le 1er met en relation `X` et `Y` s'il existe un `Z` tel que `X` est parent de `Z` et `Z` est parent de `Y`. Les deux autres mettent en relation `X` et `Y`, si en plus de ces conditions, `X` est un mâle (pour le 2e) ou une femelle (pour le 3e).

Voici quelques requêtes bâties autour de ces prédicats :

- ```
  ?- grand_pere(X, Y). % ENTRÉE
  X = marvin, Y = stan ; % ESPACE
  X = marvin, Y = shelly ; % ESPACE
  false.
  ```

  Ceci donne toutes les spécialisations de `X` et `Y` telles que `X` est un grand-père de `Y`.

- ```
  ?- grand_mere(X, stan). % ENTRÉE
  X = grandma ; % ESPACE
  false.
  ```

  Ceci donne toutes les spécialisations de `X` telles que `X` est une grand-mère de Stan.

Exemples

Ajoutons au programme précédent les trois prédicats

```
fratrie(X, Y) :-
    parent(Z, X),
    parent(Z, Y),
    X \= Y.
```

```
frere(X, Y) :-
    male(X),
    fratrie(X, Y).
```

```
soeur(X, Y) :-
    femelle(X),
    fratrie(X, Y).
```

Le 1er met en relation `X` et `Y` s'il existe un `Z` tel que `Z` est parent de `X` et `Z` est parent de `Y` avec `X` et `Y` différents. Les deux autres mettent en relation `X` et `Y`, si en plus de ces conditions, `X` est un mâle (pour le 2e) ou une femelle (pour le 3e).

Voici quelques requêtes bâties autour de ces prédicats :

- ☐
```
?- frere(X, Y). % ENTRÉE
X = randy, Y = jimbo ; % ESPACE
X = jimbo, Y = randy ; % ESPACE
X = stan, Y = shelly ; % ESPACE
X = stan, Y = shelly.
```
Ceci donne toutes les spécialisations de `X` et `Y` telles que `X` est un frère de `Y`.
Il y a une symétrie des solutions lorsque ces spécialisations portent sur deux mâles.
La dernière solution est dupliquée.

- ☐
```
?- frere(X, Y), soeur(Y, X). % ENTRÉE
X = stan, Y = shelly ; % ESPACE
X = stan, Y = shelly ; % ESPACE
X = stan, Y = shelly ; % ESPACE
X = stan, Y = shelly.
```
Ceci donne toutes les spécialisations de `X` et `Y` telles que `X` est frère de `Y` et `Y` est sœur de `X`.
Quatre solutions égales sont construites.

Exemples

Ajoutons au programme précédent le prédicat

```
oncle_ou_tante(X, Y) :-
    frere(X, Z),
    parent(Z, Y).
oncle_ou_tante(X, Y) :-
    soeur(X, Z),
    parent(Z, Y).
```

Ce prédicat met `X` et `Y` en relation s'il existe `Z` tel que `X` est frère de `Z` et `Z` est parent de `Y` ou s'il existe `Z` tel que `X` est sœur de `Z` et `Z` est parent de `Y`.

Voici quelques requêtes bâties autour de ce prédicat :

-
```
?- oncle_ou_tante(X, shelly). % ENTRÉE
X = jimbo ; % ESPACE
false.
```
Ceci donne tous les spécialisations de `X` telles que `X` est oncle ou tante de Shelly.

-
```
?- oncle_ou_tante(X, Y). % ENTRÉE
X = jimbo, Y = stan ; % ESPACE
X = jimbo, Y = shelly ; % ESPACE
false.
```
Ceci donne toutes les spécialisations de `X` et `Y` telles que `X` est oncle ou tante de `Y`.

Exemples

Ajoutons au programme précédent le prédicat

```
ancetre(X, Y) :-
    parent(X, Y).
ancetre(X, Y) :-
    parent(X, Z),
    ancetre(Z, Y).
```

Ce prédicat récursif met `X` et `Y` en relation si `X` est parent de `Y` ou s'il existe `Z` tel que `X` est parent de `Z` et `Z` est récursivement en relation avec `Y`.
Ceci revient à dire que `X` est ancêtre de `Y`.

Voici quelques requêtes bâties autour de ce prédicat :

- ```
  ?- ancetre(X, shelly). % ENTRÉE
  X = randy ; % ESPACE
  X = sharon ; % ESPACE
  X = marvin ; % ESPACE
  X = grandma ; % ESPACE
  false.
  ```

  Ceci donne toutes les spécialisations de `X` telles que `X` est ancêtre de Shelly.

- ```
  ?- ancetre(sharon, X). % ENTRÉE
  X = stan ; % ESPACE
  X = shelly ; % ESPACE
  false.
  ```

  Ceci, à l'inverse, donne toutes les spécialisations de `X` qui sont descendants de Sharon.

- ```
  ?- ancetre(X, Y), X = randy. % ENTRÉE
  X = randy, Y = stan ; % ESPACE
  X = randy, Y = shelly ; % ESPACE
  false.
  ```

  Cette requête est équivalente à la requête `ancetre(X, Y), randy = X.` ou encore à `ancetre(randy, Y).`.
  ```

Exemples

Ajoutons au programme précédent le prédicat

```
famille(X, Y) :-
    ancetre(X, Y).
famille(X, Y) :-
    ancetre(Y, X).
famille(X, Y) :-
    ancetre(Z, X),
    ancetre(Z, Y).
```

```
famille(X, Y) :-
    ancetre(X, Z),
    ancetre(Y, Z).
```

Ce prédicat **récursif** met `X` et `Y` en relation si `X` est ancêtre de `Y` ou si `Y` est ancêtre de `X` ou s'il existe `Z` tel que `Z` est ancêtre commun à `X` et à `Y`. Ceci revient à dire que `X` et `Y` appartiennent à la même famille.

Voici une requête bâtie autour de ce prédicat :

```
?- famille(sharon, X). % ENTRÉE
X = stan ; % ESPACE
X = shelly ; % ESPACE
X = grandma ; % ESPACE
X = sharon ; % ESPACE
X = stan ; % ESPACE
X = shelly ; % ESPACE
X = randy ; % ESPACE
```

```
X = sharon ; % ESPACE
X = marvin ; % ESPACE
X = grandma ; % ESPACE
X = randy ; % ESPACE
X = sharon ; % ESPACE
X = marvin ; % ESPACE
X = grandma ; % ESPACE
false.
```

Ceci donne tous les spécialisations de `X` telles que Sharon appartient à la même famille que `X`.
Observons certaines solutions dupliquées.

Comme l'arbre généalogique modélisé ne contient qu'une seule composante connexe, tout le monde est de la même famille.

Exercice : ajouter la familles des Broflovski, Cartman et McCormick, puis expérimenter un peu.

4.1.3. Prédicats

Pour tout $n \geqslant 1$, une *relation n-aire* entre des ensembles E_1, ..., E_n est un élément de $\mathcal{P}(E_1 \times \cdots \times E_n)$. L'*arité* de $\mathcal{R}$ est n.

Ainsi, une relation n-aire $\mathcal{R}$ entre E_1, ..., E_n est un ensemble de n-uplets $(x_1, \ldots, x_n)$ tels que $x_i \in E_i$ pour tout $1 \leqslant i \leqslant n$.

Si $(x_1, \ldots, x_n) \in \mathcal{R}$, nous disons que $\mathcal{R}$ *met en relation* x_1, ..., x_n. Cette propriété est notée par $\mathcal{R}(x_1, \ldots, x_n)$. Le fait que cette propriété est fausse est noté par $\not\mathcal{R}(x_1, \ldots, x_n)$.

Exemples

- Soit la relation $\mathcal{R}$ d'arité 3 entre $\mathbb{Z}$, $\mathbb{Z}$ et $\mathbb{Z}$ telle que $\mathcal{R}(z_1, z_2, z_3)$ si $z_1 - z_2 + z_3 = 0$.
  Par exemple, nous avons $\mathcal{R}(0, 0, 0)$ et $\mathcal{R}(4, -10, 6)$.

- Soit la relation $\mathcal{R}$ d'arité 3 entre l'ensemble des systèmes d'exploitation, l'ensemble des logiciels et l'ensemble des versions telle que $\mathcal{R}(s, l, v)$ si le logiciel l existe en version v sur le système d'exploitation s.
  Par exemple, nous avons

$$\mathcal{R}(\text{Manjaro Linux 24.1.2}, \text{swi-prolog}, 9.2.7), \quad \mathcal{R}(\text{Manjaro Linux 24.1.2}, \text{ocaml}, 5.2.0),$$
$$\mathcal{R}(\text{Manjaro Linux 24.1.2}, \text{i3}, 4.23).$$

Les relations n-aires sont des **généralisations des fonctions**.

En effet, une fonction $f : E_1 \times \cdots \times E_n \to E'_1 \times \cdots \times E'_{n'}$ peut se représenter la relation $\mathcal{R}_f$ d'arité $n + n'$ telle que $\mathcal{R}_f(x_1, \ldots, x_n, y_1, \ldots, y_{n'})$ si $f(x_1, \ldots, x_n) = (y_1, \ldots, y_{n'})$.

Exemples

- La fonction $f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \times \mathbb{Z}$ définie par $f(z_1, z_2) := (z_2, z_1)$ est représentée par la relation $\mathcal{R}_f$ d'arité 4 vérifiant $\mathcal{R}_f(z_1, z_2, z_2, z_1)$.
- La fonction $+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ définie par $+(z_1, z_2) := z_1 + z_2$ est représentée par la relation $\mathcal{R}_+$ d'arité 3 vérifiant $\mathcal{R}_+(z_1, z_2, z_1 + z_2)$.
- La fonction factorielle $! : \mathbb{N} \to \mathbb{N}$ est représentée par la relation $\mathcal{R}_!$ d'arité 2 vérifiant $\mathcal{R}_!(n, n!)$. Par exemple, $\mathcal{R}_!(4, 24)$.

Les fonctions permettent de définir des **transformations** de valeurs en entrée vers des valeurs en sortie.

Une relation n-aire permet d'établir une **propriété** que doivent vérifier des valeurs. Comme expliqué ici, ceci est plus général. En effet, dans ce contexte, un élément pourrait avoir plusieurs « images » par le biais d'une relation.

On appelle *prédicat* toute relation n-aire.

Un *système logique* est un cadre formel destiné à former des énoncés et à les démontrer.

Il est basé sur trois notions :

1. la **syntaxe**, qui explique comment former des énoncés ;
2. la **sémantique**, qui explique comment donner du sens aux énoncés ;
3. la **théorie de la démonstration**, qui explique comment démontrer qu'un énoncé est vrai.

En Prolog,

1. les énoncés sont les **requêtes** ;
2. les **définitions de prédicats** donnent du sens aux requêtes ;
3. la théorie de la démonstration est basée sur la **résolution**.

Un système logique doit vérifier deux caractéristiques importantes :

1. la **correction**, qui dit que tout ce qui est démontrable dans le système logique est vrai ;
2. la **complétude**, qui dit que tout ce qui est exprimable dans le système logique et qui est vrai y est démontrable.

La *logique du 1er ordre* est le système logique sur lequel PROLOG se base.

Une *formule* est définie récursivement comme étant

- un **terme** t ;
- la **négation** $\neg F$ d'une formule F ;
- la **disjonction** $F \vee F'$ de deux formules F et F' ;
- la **conjonction** $F \wedge F'$ de deux formules F et F' ;
- l'**implication** $F \to F'$ de deux formules F et F' ;
- la **quantification universelle** $\forall x\, F$ où x est une variable et F est une formule ;
- la **quantification existentielle** $\exists x\, F$ où x est une variable et F est une formule.

Exemple

La formule

$$\forall x \forall y \left((\texttt{add}(x, 1, y) \wedge \texttt{pair}(y)) \to \texttt{impair}(y) \right)$$

est formée sur des termes utilisant les prédicats $\texttt{pair}/1$, $\texttt{add}/3$ et $\texttt{impair}/1$.

PROLOG n'utilise pas cette syntaxe de la logique du 1er ordre directement.

Une **requête** `t1, ..., tn.` spécifie la formule en conjonction `t1` $\wedge \cdots \wedge$ `tn`.

Toute **variable dans une requête** est quantifiée **existentiellement**.

Exemple

En reprenant l'exemple de l'arbre généalogique, la requête `pere(marvin, X), pere(X, Y).` spécifie la formule $\exists X \, \exists Y \, \texttt{pere}(\texttt{marvin}, X) \wedge \texttt{pere}(X, Y)$

Toute **variable située dans la tête d'une clause** est quantifiée **universellement**.

Toute **variable située dans le corps d'une clause** et qui n'apparaît pas dans sa tête est quantifiée **existentiellement**.

Exemple

La clause `grand_parent(X, Y) :- parent(X, Z), parent(Z, Y).` spécifie la formule

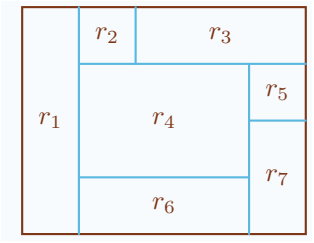
$$\forall X \, \forall Y \, ((\exists Z \, \texttt{parent}(X, Z) \wedge \texttt{parent}(Z, Y)) \rightarrow \texttt{grand\_parent}(X, Y)).$$

Nous reviendrons sur cette forme de formules logiques un peu plus loin.

4.1.4. Exemples complets

Le célèbre **théorème des quatre couleurs** annonce que toute carte peut se colorier avec quatre couleurs ou moins, sans que deux régions ayant une frontière commune ne se touchent.

Appliquons ce résultat à la carte suivante afin de trouver ses colorations autorisées :



Cette carte est constituée des sept régions r_1, ..., r_7.

Les **contraintes** de coloration font que si r_i et r_j sont adjacentes, alors la couleur de r_i doit être différente de la couleur de r_j.

Appelons `c1`, `c2`, `c3` et `c4` nos quatre couleurs.

Nous commençons par définir le prédicat `couleur/1` tel que `couleur(C)` si `C` est une couleur. Il se définit par

```
couleur(c1).
couleur(c2).
couleur(c3).
couleur(c4).
```

Exemples

Bien entendu, nous avons

```
?- couleur(c2).
true.

?- couleur(c3).
true.

?- couleur(miaou).
false.
```

L'intérêt principal réside dans le fait que le prédicat `couleur/1` est un **générateur** des couleurs autorisées.

Exemple

```
?- couleur(C).
C = c1 ;
C = c2 ;
C = c3 ;
C = c4.
```

Soit `couleurs_compatibles/2` le prédicat tel que `couleurs_compatibles(C1, C2)` si `C1` et `C2` sont deux couleurs compatibles (c'est-à-dire, différentes). Il se définit par

```
couleurs_compatibles(C1, C2) :-
    couleur(C1),
    couleur(C2),
    C1 \= C2.
```

Exemples

Voici quelques utilisations directes :

```
?- couleurs_compatibles(c1, c2).
true.

?- couleurs_compatibles(c1, c1).
false.

?- couleurs_compatibles(c2, miaou).
false.
```

Exemple

Voici une utilisation comme générateur :

```
?- couleurs_compatibles(C1, C2).
C1 = c1, C2 = c2 ;
C1 = c1, C2 = c3 ;
C1 = c1, C2 = c4 ;
C1 = c2, C2 = c1 ;
C1 = c2, C2 = c3 ;
C1 = c2, C2 = c4 ;
C1 = c3, C2 = c1 ;
C1 = c3, C2 = c2 ;
C1 = c3, C2 = c4 ;
C1 = c4, C2 = c1 ;
C1 = c4, C2 = c2 ;
C1 = c4, C2 = c3 ;
false.
```

Soit `coloration/7` le prédicat tel que `coloration(C1, C2, C3, C4, C5, C6, C7)` si la coloration de la carte initiale obtenu en colorant r_1 de la couleur `C1`, r_2 de la couleur `C2`, ..., r_7 de la couleur `C7` est compatible. Il se définit par

```
coloration(C1, C2, C3, C4, C5, C6, C7) :-
    couleurs_compatibles(C1, C2),
    couleurs_compatibles(C1, C4),
    couleurs_compatibles(C1, C6),
    couleurs_compatibles(C2, C3),
    couleurs_compatibles(C2, C4),
    couleurs_compatibles(C3, C4),
    couleurs_compatibles(C3, C5),
    couleurs_compatibles(C4, C5),
    couleurs_compatibles(C4, C6),
    couleurs_compatibles(C4, C7),
    couleurs_compatibles(C5, C7),
    couleurs_compatibles(C6, C7).
```

Ce prédicat s'obtient en considérant toutes les régions r_i et r_j de la carte qui ont une frontière commune et en ajoutant la condition que les couleurs `Ci` et `Cj` doivent être compatibles. Cela se fait, avec les définitions précédentes, par `couleurs_compatibles(Ci, Cj)`.

Les solutions sont

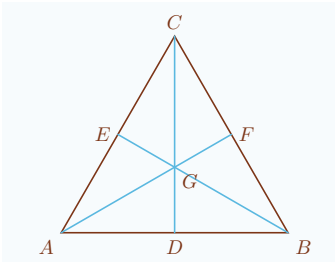
```
?- coloration(C1, C2, C3, C4, C5, C6, C7).
C1 = C3, C3 = C7, C7 = c1, C2 = C5, C5 = C6, C6 = c2, C4 = c3 ;
C1 = C3, C3 = c1, C2 = C5, C5 = C6, C6 = c2, C4 = c3, C7 = c4 ;
C1 = C3, C3 = C7, C7 = c1, C2 = C6, C6 = c2, C4 = c3, C5 = c4 ;
C1 = C5, C5 = c1, C2 = C6, C6 = c2, C3 = C7, C7 = c4, C4 = c3 ;
...
C1 = C3, C3 = c4, C2 = C5, C5 = C6, C6 = c3, C4 = c2, C7 = c1 ;
C1 = C3, C3 = C7, C7 = c4, C2 = C5, C5 = C6, C6 = c3, C4 = c2 ;
false.
```

Il y en a beaucoup.

Exercice : expérimenter avec d'autres cartes (en mettant à jour `coloration/7` de sorte à tenir compte des nouvelles frontières).

Nous verrons plus tard comment **collecter dans une liste** toutes les solutions calculées. Ceci nous sera utile pour, par exemple dénombrer les solutions ou encore pour réaliser un traitement supplémentaire sur ces dernières.

Intéressons-nous à un problème de dénombrement concret.



Voici une figure géométrique.

Nous souhaitons **dénombrer les triangles** qui la composent.

Par exemple, ABC, ABG et BCE sont de tels triangles.

Notre 1re tâche consiste à représenter cette figure. Observons que les coordonnées exactes des points n'importent pas pour notre objectif. Seule la **structure de la figure intervient**.

Nous allons, pour la représenter, utiliser

- un atome par point : `a`, `b`, `c`, `d`, `e`, `f` et `g` ;
- un atome par segment maximal : `ab`, `ac`, `af`, `bc`, `be` et `cd` ;
- un prédicat qui teste si un segment contient un point.

Soit `segment_point/2` le prédicat tel que `segment_point(SEG, P)` si le segment `SEG` contient le point `P`. Il se définit par

```
segment_point(ab, a).
segment_point(ab, b).
segment_point(ab, d).
segment_point(ac, a).
segment_point(ac, c).
segment_point(ac, e).
segment_point(af, a).
segment_point(af, f).
segment_point(af, g).
segment_point(bc, b).
segment_point(bc, c).
segment_point(bc, f).
segment_point(be, b).
segment_point(be, e).
segment_point(be, g).
segment_point(cd, c).
segment_point(cd, d).
segment_point(cd, g).
```

Cet ensemble de faits représente notre figure « en dur ».

Ce prédicat n'est pas générique. Pour adapter notre solution à d'autres figures, il faudra mettre à jour ce prédicat.

Nous allons mettre au point des prédicats `triangle_vN/3` qui sont tels que `triangle_vN(P1, P2, P3)` si les trois points `P1`, `P2` et `P3` forment un triangle dans la figure.

La requête `triangle_vN(P1, P2, P3).` va nous permettre d'**engendrer toutes les solutions**, qu'il nous suffira de compter (comme mentionné avant, il existe des méthodes automatiques pour compter les solutions, chose qui sera vue par la suite).

Nous donnerons plusieurs versions que nous affinerons progressivement,

Voici une première tentative :

```
triangle_v1(P1, P2, P3) :-
    segment_point(P1_P2, P1),
    segment_point(P1_P2, P2),
    segment_point(P2_P3, P2),
    segment_point(P2_P3, P3),
    segment_point(P1_P3, P1),
    segment_point(P1_P3, P3).
```

Ceci est très naturel : nous affirmons que pour avoir un triangle formé par les points `P1`, `P2` et `P3`, nous devons avoir trois segments `P1_P2`, `P2_P3` et `P1_P3` contenant les points de manière cohérente.

Malheureusement,

```
?- triangle_v1(P1, P2, P3).
P1 = P2, P2 = P3, P3 = a ;
...
```

construit une solution formée d'un **triangle réduit à un point**.

Améliorons notre proposition en demandant que les trois points soient différents. Cela donne

```
triangle_v2(P1, P2, P3) :-
    segment_point(P1_P2, P1),
    segment_point(P1_P2, P2),
    segment_point(P2_P3, P2),
    segment_point(P2_P3, P3),
    segment_point(P1_P3, P1),
    segment_point(P1_P3, P3),
    P1 \= P2,
    P2 \= P3,
    P3 \= P1.
```

Ceci est très naturel : en plus des conditions précédentes, nous demandons que les trois points `P1`, `P2` et `P3` soient deux à deux différents.

Malheureusement,

```
?- triangle_v2(P1, P2, P3).
P1 = a, P2 = b, P3 = d
...
```

construit une solution formée d'un **triangle dont les points sont alignés**.

Améliorons notre proposition en demandant de plus que les trois segments soient différents. Cela donne

```
triangle_v3(P1, P2, P3) :-
    segment_point(P1_P2, P1),
    segment_point(P1_P2, P2),
    segment_point(P2_P3, P2),
    segment_point(P2_P3, P3),
    segment_point(P1_P3, P1),
    segment_point(P1_P3, P3),
    P1 \= P2,
    P2 \= P3,
    P3 \= P1,
    P1_P2 \= P2_P3,
    P2_P3 \= P1_P3,
    P1_P3 \= P1_P2.
```

Ceci est très naturel : en plus des conditions précédentes, nous demandons que les trois segments `P1_P2`, `P1_P3` et `P2_P3` soient deux à deux différents.

Malheureusement,

```
?- triangle_v3(P1, P2, P3).
P1 = a, P2 = b, P3 = c ;
...
P1 = b, P2 = a, P3 = c ;
...
```

construit des solutions qui représentent **le même triangle**. Ceci n'est pas acceptable pour la problématique de dénombrement.

Pour éviter les solutions qui correspondent à des mêmes triangles, adoptons la convention que nos points sont totalement ordonnés selon l'ordre alphabétique et que tout triangle est spécifié par ses trois points dans le bon ordre.

Par exemple, les trois points `a`, `d` et `g`, dans cet ordre, forment un triangle. À l'inverse, les trois points `d`, `g`, `a`, dans cet ordre, ne forment pas un triangle.

Soit `inf/2` le prédicat tel que `inf(P1, P2)` si le point `P1` est inférieur pour l'ordre alphabétique au point `P2`.

Pour le définir, utilisons le prédicat `suivant/2` tel que `suivant(P1, P2)` si le point `P1` est suivi par le point `P2` dans l'alphabet.

Voici leurs définitions :

```
suivant(a, b).
suivant(b, c).
suivant(c, d).
suivant(d, e).
suivant(e, f).
suivant(f, g).
suivant(g, h).
```

```
inf(P, P).
inf(P1, P2) :-
    suivant(P1, P3),
    inf(P3, P2).
```

Le second prédicat est récursif.

Voici notre prédicat pourvu de l'amélioration précédente, avec la requête de génération des solutions :

```
triangle_v4(P1, P2, P3) :-
    segment_point(P1_P2, P1),
    segment_point(P1_P2, P2),
    segment_point(P2_P3, P2),
    segment_point(P2_P3, P3),
    segment_point(P1_P3, P1),
    segment_point(P1_P3, P3),
    P1 \= P2,
    P2 \= P3,
    P3 \= P1,
    P1_P2 \= P2_P3,
    P2_P3 \= P1_P3,
    P1_P3 \= P1_P2,
    inf(P1, P2),
    inf(P2, P3).
```

```
?- triangle_v4(P1, P2, P3).
P1 = a, P2 = b, P3 = c ;
P1 = a, P2 = b, P3 = f ;
P1 = a, P2 = b, P3 = e ;
P1 = a, P2 = b, P3 = g ;
P1 = a, P2 = d, P3 = g ;
P1 = b, P2 = d, P3 = g ;
P1 = a, P2 = c, P3 = f ;
P1 = a, P2 = c, P3 = d ;
P1 = a, P2 = c, P3 = g ;
P1 = a, P2 = e, P3 = g ;
P1 = c, P2 = e, P3 = g ;
P1 = b, P2 = c, P3 = e ;
P1 = b, P2 = c, P3 = d ;
P1 = b, P2 = c, P3 = g ;
P1 = b, P2 = f, P3 = g ;
P1 = c, P2 = f, P3 = g ;
false.
```

La réponse au problème initial est donc 16.

Exercice : expérimenter avec d'autres figures géométriques (en mettant à jour `segment_point/2` et `suivant/2` adéquatement).

Nous allons programmer ici une petite bibliothèque de manipulation des **entiers naturels**.

Nous allons considérer l'atome `z` qui représente l'entier 0 et l'atome `s` qui permet de construire les autres entiers comme successeurs itérés de 0.

Exemples

- Le terme `s(z)` représente l'entier 1.
- Le terme `s(s(z))` représente l'entier 2.
- Le terme `s(s(s(s(z))))` représente l'entier 4.

Soit `entier/1` le prédicat tel que `entier(N)` si `N` est un entier. Il se définit par

```
entier(z).
entier(s(N)) :-
    entier(N).
```

Exemples

```
?- entier(s(s(z))).
true.
```

```
?- entier(miaou).
false.
```

```
?- entier(s(s(miaou))).
false.
```

Soit `inf/2` le prédicat tel que `inf(N1, N2)` si l'entier `N1` **est inférieur** à l'entier `N2`. Il se définit par

```
inf(z, _).
inf(s(N1), s(N2)) :-
    inf(N1, N2).
```

Le `_` se comporte comme un **joker** (mettre une variable ici ferait qu'elle serait non utilisée dans le corps de la clause, ce qui provoquerait un avertissement de « variable singleton »).

Exemples

```
?- inf(z, s(s(z))).
true.
```

```
?- inf(s(z), s(z)).
true.
```

```
?- inf(miaou, s(z)).
false.
```

De façon similaire, nous pouvons définir le prédicat `infs/2` tel que `infs(N1, N2)` si `N1` est strictement inférieur à `N2`.

Il est aussi possible de définir les prédicats duaux `sup/2` et `sups/2`.

Exercice : implanter ces trois prédicats supplémentaires.

Une problématique se pose lorsque nous souhaitons **manipuler des nombres concrets** : leur construction comme nombres de Peano est de la taille du nombre lui-même, ce qui peut être peu maniable.

Pour répondre à cela, il est intéressant d'avoir des prédicats `entier_n/1` tels que `entier_n(N)` si `N` est l'entier `n`. Nous pouvons alors écrire

```
entier_n(N), t1, ..., tk
```

où `t1, ..., tk` est une conjonction de termes. La partie `entier_n(N)` fait que la variable `N` est spécialisée à la valeur qui rend `entier_n(N)` vrai, et donc à la valeur `n` dans `t1, ..., tk`.

Une implantation possible des premiers prédicats de cette forme est

```
entier_0(z).
entier_1(s(X)) :- entier_0(X).
```

```
entier_2(s(X)) :- entier_1(X).
entier_3(s(X)) :- entier_2(X).
```

```
entier_4(s(X)) :- entier_3(X).
...
```

Exemples

```
?- entier_1(N1), entier_3(N2), inf(N1, N2).
N1 = s(z), N2 = s(s(s(z))).
```

Les deux premières parties de la requête font que `N1` (resp. `N2`), devant vérifier le prédicat `entier_1/1` (resp. `entier_2/1`), doit se spécialiser en `s(z)` (resp. `s(s(z))`). Ainsi, dans `inf(N1, N2)` les variables `N1` et `N2` sont spécialisées en les valeurs voulues.

Soit `add/3` le prédicat tel que `add(N1, N2, N3)` si l'entier `N1` **additionné** à l'entier `N2` est égal à l'entier `N3`. Il se définit par

```
add(z, N, N) :-
    entier(N).
add(s(N1), N2, s(N3)) :-
    add(N1, N2, N3).
```

Exemple

```
?- entier_4(N1), entier_2(N2), add(N1, N2, X).
N1 = s(s(s(s(z)))), N2 = s(s(z)), X = s(s(s(s(s(s(z)))))).
```

Le prédicat `add/3` est utilisé ici pour calculer le résultat d'une addition.

Exemple

```
?- entier_3(N), add(X1, X2, N).
N = X2, X2 = s(s(z)), X1 = z ;
N = s(s(z)), X1 = s(z), X2 = s(s(z)) ;
N = s(s(z)), X1 = s(s(z)), X2 = s(z) ;
N = X1, X1 = s(s(z)), X2 = z ;
false.
```

Le prédicat `add/3` est utilisé différemment : comme il n'y a pas de notion d'entrée et de sortie pour un prédicat, nous spécialisons son dernier argument et laissons les deux 1ers variables.
Ceci permet de rechercher toutes les manières de décomposer 3 comme somme de deux entiers naturels.

Soit `mul/3` le prédicat tel que `mul(N1, N2, N3)` si l'entier `N1` **multiplié** à l'entier `N2` est égal à l'entier `N3`. Il se définit par

```
mul(z, N, z) :-
    entier(N).
mul(s(N1), N2, N3) :-
    mul(N1, N2, N4),
    add(N4, N2, N3).
```

Exemple

```
?- entier_3(N1), entier_2(N2), mul(N1, N2, R).
N1 = s(s(s(z))), N2 = s(s(z)), R = s(s(s(s(s(z))))).
```

Le prédicat `mul/3` est utilisé ici pour calculer le résultat d'une multiplication.

Exemple

```
?- entier_6(N), inf(X1, N), inf(X2, N), mul(X1, X2, N).
N = X2, X2 = s(s(s(s(s(z))))), X1 = s(z) ;
N = s(s(s(s(s(z))))), X1 = s(s(z)), X2 = s(s(s(z))) ;
N = s(s(s(s(s(z))))), X1 = s(s(s(z))), X2 = s(s(z)) ;
N = X1, X1 = s(s(s(s(s(z))))), X2 = s(z) ;
false.
```

Ceci recherche les factorisations de 4.
Dans la requête, nous forçons `X1` et `X2` à être inférieurs à `N`.
Exercice : observer ce qu'il se passe sans ces contraintes.

Soit `fact/2` le prédicat tel que `fact(N1, N2)` si la **factorielle** de l'entier `N1` est l'entier `N2`. Il se définit par

```
fact(z, s(z)).
fact(s(N1), N2) :-
    fact(N1, N3),
    mul(s(N1), N3, N2).
```

Exemple

```
?- entier_3(N), fact(N, X).
N = s(s(s(z))), X = s(s(s(s(s(z))))).
```

Le prédicat `fact/2` est utilisé ici pour calculer le résultat d'une factorielle.

Exemple

```
?- entier_5(N), inf(X, N), fact(X, N).
false.

?- entier_6(N), inf(X, N), fact(X, N).
N = s(s(s(s(s(z))))), X = s(s(s(z))) ;
false.
```

Ceci recherche les antécédents de 5 et de 6 pour la factorielle.
Dans les requête, nous forçons `X` à être inférieur à `N`, pour les mêmes raisons que dans le 2e exemple de la page précédente.

Soit `fibo/2` le prédicat tel que `fibo(N1, N2)` si le terme en position `N1` de la **suite de Fibonacci** est l'entier `N2`. Il se définit par

```
fibo(z, s(z)).
fibo(s(z), s(z)).
fibo(s(s(N1)), N2) :-
    fibo(s(N1), N3),
    fibo(N1, N4),
    add(N4, N3, N2).
```

Exemple

```
entier_5(N), fibo(N, X).
N = s(s(s(s(s(z))))), X = s(s(s(s(s(s(s(s)))))))).
```

Le prédicat `fibo/2` est utilisé ici pour calculer le résultat de la fonction de Fibonacci.

Exemple

```
?- entier_6(N), inf(X, N), fibo(X, N).
false.

?- entier_5(N), inf(X, N), fibo(X, N).
N = s(s(s(s(z)))))), X = s(s(s(z)))) ;
false.
```

Ceci recherche les antécédents de 5 et de 6 pour la fonction Fibonacci.
Dans les requête, nous forçons `X` à être inférieur à `N`, pour les mêmes raisons que dans le 2e exemple de la page précédente.

4.2. Bases théoriques

4.2.1. Unification

Une pièce fondamentale intervenant dans la **résolution** des requêtes est l'**unification de termes**.

Intuitivement, *unifier* deux termes t_1 et t_2 consiste à trouver une manière de remplacer simultanément les variables de t_1 et de t_2 par des termes de sorte à les rendre égaux.

Exemple

Considérons les deux termes $t_1 := \mathtt{add}(X, Y, \mathtt{z})$ et $t_2 := \mathtt{add}(\mathtt{s}(\mathtt{z}), Z, T)$.

Pour les rendre égaux, il faut identifier

- X et $\mathtt{s}(\mathtt{z})$;
- Y et Z ;
- $\mathtt{z}$ et T.

En remplaçant dans t_1 et t_2, X par $\mathtt{s}(\mathtt{z})$, Y par Z et T par $\mathtt{z}$, nous obtenons le même terme $t_2 := \mathtt{add}(\mathtt{s}(\mathtt{z}), Z, \mathtt{z})$.

Exemple

Considérons les deux termes $t_1 := \mathtt{inf}(\mathtt{z}, \mathtt{s}(X))$ et $t_2 := \mathtt{inf}(\mathtt{s}(Y), Z)$.

Ces deux termes ne sont pas unifiables car il faudrait identifier $\mathtt{z}$ et $\mathtt{s}(X)$, ce qui est impossible.

En effet, ces deux sous-termes ont comme racines des atomes différents.

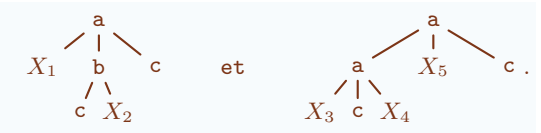
Ceci revient à résoudre une **équation sur les termes** dont les **variables sont les inconnues**.

Il est possible de visualiser comment deux termes peuvent s'unifier en tentant de superposer leurs représentations sous forme d'arbres.

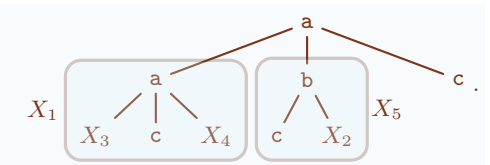
Il faut que tout atome se superpose soit sur une variable, soit sur un atome identique.

Exemple

Soient les termes $\mathrm{t}_1 := \mathrm{a}(X_1, \mathrm{b}(\mathrm{c}, X_2), \mathrm{c})$ et $\mathrm{t}_2 := \mathrm{a}(\mathrm{a}(X_3, \mathrm{c}, X_4), X_5, \mathrm{c})$.
Leurs représentations graphiques respectives sont



La superposition de ces termes donne



Les termes t_1 et t_2 sont donc unifiables, car, en remplaçant dans ces deux termes X_1 par $\mathrm{a}(X_3, \mathrm{c}, X_4)$ et X_5 par $\mathrm{b}(\mathrm{c}, X_2)$, nous obtenons le même terme.

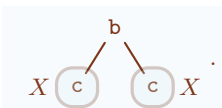
L'exemple précédent était facile à traiter car les deux termes à unifier ne possédaient pas de variable en commun.

Voici quelques exemples où cela arrive.

Exemple

Soient les termes $\mathrm{t}_1 := \mathrm{b}(X, \mathrm{c})$ et $\mathrm{t}_2 := \mathrm{b}(\mathrm{c}, X)$.
Leurs représentations graphiques respectives sont

La superposition de ces termes donne

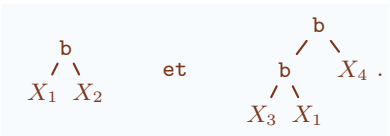


Les termes t_1 et t_2 sont donc unifiables, car, en remplaçant dans ces deux termes X par c et X par c (cette répétition est volontairement mentionnée), nous obtenons le même terme.

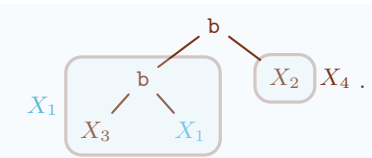
Exemple

Soient les termes $t_1 := b(X_1, X_2)$ et $t_2 := b(b(X_3, X_1), X_4)$.

Leurs représentations graphiques respectives sont



La superposition de ces termes donne



Pour unifier t_1 et t_2, il faut donc unifier X_1 et $b(X_3, X_1)$. Comme ce dernier terme contient la variable X_1, ceci est impossible.

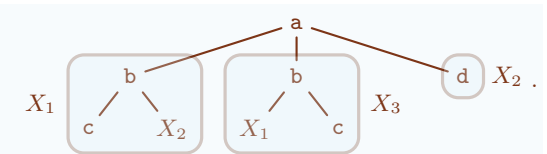
Les termes t_1 et t_2 ne sont donc pas unifiables.

Voici un exemple un peu plus complexe.

Exemple

Soient les termes $t_1 := a(X_1, b(X_1, c), X_2)$ et $t_2 := a(b(c, X_2), X_3, d)$.
Leurs représentations graphiques respectives sont

La superposition de ces termes donne



Pour unifier t_1 et t_2, il faut donc unifier

- X_1 avec $b(c, X_2)$;
- X_3 avec $b(X_1, c)$;
- X_2 avec d.

Les termes t_1 et t_2 sont donc unifiables.

En simplifiant, ceci revient à unifier

- X_1 avec $b(c, d)$;
- X_3 avec $b(b(c, d), c)$;
- X_2 avec d.

Mettons un peu de formalisme à tout cela.

Un *substitution* est une fonction σ dont le domaine est l'ensemble des variables et le codomaine est l'ensemble des termes.

Si t est un terme, l'*application* de σ à t est le terme $t \cdot \sigma$ obtenu en remplaçant simultanément, pour toute variable X, toutes les occurrences de X dans t par $\sigma(X)$.

Exemples

- Soit σ la substitution telle que $\sigma(X) = g(Y)$. Nous avons $f(X) \cdot \sigma = f(g(Y))$.
- Soit σ la substitution telle que $\sigma(X_1) = b(d, d)$, $\sigma(X_2) = a(c, X_4, X_4)$ et $\sigma(X_3) = X_4$. Nous avons

$$a(X_1, X_2, b(X_1, X_3)) \cdot \sigma = a(b(d, d), a(c, X_4, X_4), b(b(d, d), X_4)).$$

Une substitution σ *unifie* deux termes t_1 et t_2 si $t_1 \cdot \sigma = t_2 \cdot \sigma$.

Exemple

Les termes $b(X, c)$ et $b(c, X)$ sont unifiés par toute substitution σ telle que $\sigma(X) = c$.

La notation pratique

$$\sigma = \{X_1 \mapsto \mathrm{t}_1, \ldots, X_n \mapsto \mathrm{t}_n\}$$

signifie que σ est la substitution vérifiant

$$\sigma(X) = \begin{cases} \mathrm{t} & \text{si } X \mapsto \mathrm{t} \in \sigma, \\ X & \text{sinon.} \end{cases}$$

Exemple

La définition

$$\sigma := \{X_1 \mapsto \mathrm{b}(X_1, X_2), \quad X_2 \mapsto \mathrm{d}, \quad X_3 \mapsto \mathrm{a}(X_4, X_5, \mathrm{b}(\mathrm{d}, X_2))\},$$

spécifie la substitution σ vérifiant $\sigma(X_1) = \mathrm{b}(X_1, X_2)$, $\sigma(X_2) = \mathrm{d}$, $\sigma(X_3) = \mathrm{a}(X_4, X_5, \mathrm{b}(\mathrm{d}, X_2))$ et $\sigma(X) = X$ pour toute variable X différente de X_1, X_2 et X_3.

La *substitution identité* ι est définie par $\iota := \emptyset$. Ainsi, pour toute variable X, $\iota(X) = X$.

La *composition* de deux substitutions σ_1 et σ_2 est la substitution $\sigma_1 \circ \sigma_2$ qui est telle que, pour toute variable X,

$$(\sigma_1 \circ \sigma_2)(X) = (X \cdot \sigma_2) \cdot \sigma_1.$$

Ainsi, l'image de la variable X par $\sigma_1 \circ \sigma_2$ se calcule en considérant le terme obtenu en appliquant σ_2 à X, puis en considérant le terme obtenu en appliquant σ_1 sur ce dernier.

Exemple

Soient les deux substitutions

$$\sigma_1 := \{X_2 \mapsto \mathtt{a}(X_1, X_2, \mathtt{d}),\ \ X_3 \mapsto \mathtt{b}(X_1, X_2)\}$$

et

$$\sigma_2 := \{X_1 \mapsto \mathtt{b}(X_2, X_3),\ \ X_2 \mapsto \mathtt{d}\}.$$

Nous avons

$$\sigma_1 \circ \sigma_2 = \{X_1 \mapsto \mathtt{b}(\mathtt{a}(X_1, X_2, \mathtt{d}), \mathtt{b}(X_1, X_2)),\ \ X_2 \mapsto \mathtt{d},\ \ X_3 \mapsto \mathtt{b}(X_1, X_2)\}.$$

Une substitution σ_1 est *plus générale* qu'une substitution σ_2 s'il existe une substitution σ_3 telle que $\sigma_2 = \sigma_3 \circ \sigma_1$. Nous disons aussi dans ce cas que σ_2 est *plus spécifique* que σ_1.

Exemple

Soient les substitutions

$$\sigma_1 := \{X_1 \mapsto X_2,\ X_2 \mapsto \mathsf{c}(X_3)\} \quad \text{et} \quad \sigma_2 := \{X_1 \mapsto \mathsf{c}(\mathsf{d}),\ X_2 \mapsto \mathsf{c}(\mathsf{e}),\ X_3 \mapsto \mathsf{e}\}.$$

En posant

$$\sigma_3 := \{X_2 \mapsto \mathsf{c}(\mathsf{d}),\ X_3 \mapsto \mathsf{e}\},$$

nous avons $\sigma_2 = \sigma_3 \circ \sigma_1$. Ainsi, σ_1 est plus générale que σ_2.

Cette notion est importante car pour **unifier** deux termes, nous cherchons la **substitution la plus générale** possible qui fait l'affaire.

Exemple

Soient $t_1 := \mathsf{b}(X_1, \mathsf{d})$ et $t_2 := \mathsf{b}(X_2, X_3)$. Nous pouvons vérifier facilement que les substitutions

$$\sigma_1 := \{X_1 \mapsto X_2,\ X_3 \mapsto \mathsf{d}\} \quad \text{et} \quad \sigma_2 := \{X_1 \mapsto \mathsf{d},\ X_2 \mapsto \mathsf{d},\ X_3 \mapsto \mathsf{d}\}$$

unifient t_1 et t_2.

Cependant, comme σ_1 est plus générale que σ_2, la substitution σ_1 est préférable.

L'**algorithme d'unification** se décrit ainsi.

Il prend deux termes t_1 et t_2 en entrée et renvoie la substitution la plus générale qui unifie t_1 et t_2 si et seulement si ces deux termes sont unifiables.

```
Fonction unifier(t1, t2)
    Si t1 et t2 sont tous deux égaux à la même variable X
        Renvoyer ι
    Sinon si (t1 est une variable qui apparaît dans t2) ou (t2 est une variable qui apparaît dans t1)
        Renvoyer Échec
    Sinon si t1 est une variable X
        Renvoyer {X ↦ t2}
    Sinon si t2 est une variable X
        Renvoyer {X ↦ t1}
    Sinon si les racines de t1 et t2 sont des atomes différents
        Renvoyer Échec
    Sinon
        Soit σ := ι
        Pour chaque couple (s1,s2) de fils en mêmes positions respectivement de t1 et t2
            Soit σ := unifier(s1 · σ, s2 · σ) ∘ σ
        Renvoyer σ
    Fin
Fin
```

Exemple

Soient les termes $t_1 := a(e(X_1), X_1, X_2)$ et $t_2 := a(X_2, c, e(X_3))$.

1. L'appel $\texttt{unifier}(t_1, t_2)$ déclenche le tout dernier cas.
2. Initialement, $\sigma := \iota$.
3. Ensuite, nous considérons les fils en 1res positions de t_1 et t_2. Ceci donne

$$\sigma := \texttt{unifier}(e(X_1) \cdot \sigma, X_2 \cdot \sigma) \circ \sigma = \texttt{unifier}(e(X_1), X_2) = \{X_2 \mapsto e(X_1)\}.$$

4. Ensuite, nous considérons les fils en 2es positions de t_1 et t_2. Ceci donne

$$\sigma := \texttt{unifier}(X_1 \cdot \sigma, c \cdot \sigma) \circ \sigma = \texttt{unifier}(X_1, c) \circ \sigma = \{X_1 \mapsto c\} \circ \sigma = \{X_1 \mapsto c,\ X_2 \mapsto e(c)\}.$$

5. Ensuite, nous considérons les fils en 3es positions de t_1 et t_2. Ceci donne

$$\sigma := \texttt{unifier}(X_2 \cdot \sigma, e(X_3) \cdot \sigma) \circ \sigma = \texttt{unifier}(e(c), e(X_3)) \circ \sigma = \{X_3 \mapsto c\} \circ \sigma$$
$$= \{X_1 \mapsto c,\ X_2 \mapsto e(c),\ X_3 \mapsto c\}.$$

Cette substitution σ est le résultat de l'appel initial.

Exercice : appliquer l'algorithme sur les termes $t_1 := a(e(X_1), X_1, X_1)$ et $t_2 := a(X_2, c, e(X_3))$.

En Prolog, il est possible de **demander explicitement une unification** via l'opérateur `=`.

Exemples

Voici quelques requêtes, directement soumises à SWI-Prolog :

```
?- a(X, Y) = a(X, t).
Y = t.
```

```
?- a(X, Y) = a(Y, t).
X = Y, Y = t.
```

```
?- a(X, Y) = b(X, Y).
false.
```

```
?- a(X, a(Z, Z)) = a(Y, Y).
X = Y, Y = a(Z, Z).
```

Exemple

Voici deux définitions équivalentes du prédicat `inf/2` sur les entiers de Peano :

```
inf(z, _).
inf(s(N1), s(N2)) :-
    inf(N1, N2).
```

```
inf(N1, _) :-
    N1 = z.
inf(N1, N2) :-
    N1 = s(P1),
    N2 = s(P2),
    inf(P1, P2).
```

Dans la version de droite, l'unification est explicite. Les deux clauses du prédicat acceptent des paramètres génériques `N1` et `N2` qui sont contraints par la suite.

Dans la version de gauche, les deux clauses possèdent des arguments qui sont d'emblée spécialisés en la forme souhaitée. Cette version est en général bien préférable.

4.2.2. Résolution

Une *clause de Horn* est une formule de la forme

$$(H_1 \wedge H_2 \wedge \cdots \wedge H_n) \rightarrow C$$

où H_1, H_2, ..., H_n et C sont des termes.

Exemples

Quelques clauses de Horn :

- $(\texttt{divise}(2, X) \wedge \texttt{divise}(3, X)) \rightarrow \texttt{divise}(6, X)$;
- $(\texttt{fainéant}(Chat) \wedge \texttt{gourmand}(Chat) \wedge \texttt{jeune}(Chat)) \rightarrow \texttt{gros}(Chat)$.

Les H_i sont les *hypothèses* (elles forment le *corps* de la clause) et C est la *conclusion* (ou encore *tête* de la clause).

Cette formule explique le fait que **si** chaque H_i est vraie, **alors** C est vraie.

La véracité de chacune des hypothèses est une **condition suffisante** pour la conclusion.

Attention : les hypothèses ne sont pas forcément des conditions nécessaires à la conclusion.

Lorsque $n = 0$, les hypothèses étant vides, C est vraie **inconditionnellement**. Dans ce cas, la clause de Horn exprime un **fait**.

Soit $q := q_1 \wedge \cdots \wedge q_n$ une requête appelée *but*. **Résoudre** ce but signifie calculer toutes les substitutions σ qui sont telles que pour tout $1 \leqslant i \leqslant n$, tous les $q_i \cdot \sigma$ sont vrais simultanément.

Pour cela, PROLOG utilise l'algorithme de *résolution linéaire sélective de clauses définies* (SLD). Expliquons-le pas à pas.

L'algorithme part du but q et tente de **justifier** chaque sous-but q_i en commençant par q_1.

Pour cela, il cherche dans le programme la 1re clause `t :- t1, ..., tk` telle que q_1 et `t` sont unifiables. Il s'agit de la clause *sélectionnée*.

Exemple

Considérons le programme

```
parent(marvin, randy).
parent(marvin, jimbo).
parent(randy, shelly).
parent(randy, stan).
grand_parent(X, Y) :- parent(X, Z), parent(Z, Y).
```

Considérons le but à résoudre

```
grand_parent(marvin, X).
```

Ce but est tout d'abord utilisé pour être unifié avec les quatre 1res clauses, ce qui échoue.

Ensuite, vient la 5e clause où une unification entre `grand_parent(marvin, X)` et `grand_parent(X1, Y1)` (obtenue en renommant ses variables de manière unique) est tentée.

La substitution résultante est $\sigma := \{X^1 \mapsto \texttt{marvin},\ Y^1 \mapsto X\}$.

Une fois cette sélection effectuée, l'algorithme **remplace** $q_1, q_2, \ldots, q_n$ par

$$\texttt{t1} \cdot \sigma, \ldots, \texttt{tk} \cdot \sigma, \quad q_2 \cdot \sigma, \ldots, q_n \cdot \sigma$$

où σ est la substitution calculée par l'unification de q_1 et de `t`.

Exemple

Depuis l'exemple précédent, le but `grand_parent(marvin, X)` est remplacé par `parent(marvin, Z), parent(Z, X)`.

Ceci donne un nouveau but, qu'il faut continuer à résoudre de la même manière. Nous obtiendrons après la sélection d'une clause une substitution σ' par l'unification du 1er sous-but et de la tête de la clause. La **substitution courante** deviendra $\sigma' \circ \sigma$.

Exemple

Depuis l'exemple précédent, le but est `parent(marvin, Z), parent(Z, X)` et la substitution en cours de calcul est $\sigma := \left\{X^1 \mapsto \texttt{marvin},\ Y^1 \mapsto X\right\}$.
La clause sélectionnée (qui est un fait ici) est `parent(marvin, randy).` et la substitution calculée par l'unification de `parent(marvin, Z)` avec la tête de ce fait est $\sigma' = \{Z \mapsto \texttt{randy}\}$.
La substitution courante est $\left\{X^1 \mapsto \texttt{marvin}, Y^1 \mapsto X, Z \mapsto \texttt{randy}\right\}$ et le but courant est `parent(randy, X)`.

Les étapes de sélection, unification, et mise à jour du but et de la substitution courante sont poursuivies jusqu'à obtenir un **but vide**, ce qui signifie un succès dans la requête initiale. La substitution courante est alors une solution à la requête initiale.

Exemple

Depuis l'exemple précédent, nous avons `parent(randy, X)` comme but courant et $\{X^1 \mapsto \texttt{marvin},\ Y^1 \mapsto X,\ Z \mapsto \texttt{randy}\}$ comme substitution courante.
La clause sélectionnée est `parent(randy, shelly).` et la substitution calculée par l'unification de `parent(randy, X)` avec la tête de cette clause est $\{X \mapsto \texttt{shelly}\}$.
La nouvelle substitution courante est $\{X^1 \mapsto \texttt{marvin},\ Y^1 \mapsto \texttt{shelly},\ Z \mapsto \texttt{randy},\ X \mapsto \texttt{shelly}\}$ et le nouveau but est vide.
Ainsi, cette dernière substitution est une solution au but initial.

Lorsqu'une substitution est un résultat, seules les **variables qui apparaissent dans le but initial** sont mentionnées.

Exemple

Depuis l'exemple précédent, $\{X \mapsto \texttt{shelly}\}$ est une solution à la requête initiale `grand_parent(marvin, X)`.

Une fois qu'une solution est trouvée, l'algorithme effectue un **retour arrière** au dernier endroit où il a sélectionné une clause de sorte à vérifier si une autre peut être sélectionnée.

Exemple

Depuis l'exemple précédent, depuis le but courant `parent(randy, X)` et la substitution courante $\{X^1 \mapsto \texttt{marvin},\ Y^1 \mapsto X,\ Z \mapsto \texttt{randy}\}$, la prochaine clause sélectionnée est `parent(randy, stan)`.

La substitution calculée par l'unification est $\{X \mapsto \texttt{stan}\}$.

La nouvelle substitution courante est $\{X^1 \mapsto \texttt{marvin},\ Y^1 \mapsto \texttt{stan},\ Z \mapsto \texttt{randy},\ X \mapsto \texttt{stan}\}$ et le nouveau but est vide.

Ainsi, $\{X \mapsto \texttt{stan}\}$ est une solution à la requête initiale `grand_parent(marvin, X)`.

L'algorithme procède en continuant de cette manière pour **rechercher toutes les solutions**.

Lorsque aucune clause ne peut être sélectionnée de sorte que sa tête puisse être unifiée avec le 1^{er} sous-but, une **situation d'échec** est rencontrée.

La recherche continue ensuite par retour arrière s'il reste des clauses à considérer jusqu'à épuisement.

L'**algorithme de résolution** se décrit ainsi.

Il prend une suite de clauses P et un but $q_1 \wedge \cdots \wedge q_n$ en entrée, et renvoie l'ensemble des solutions de ce but.

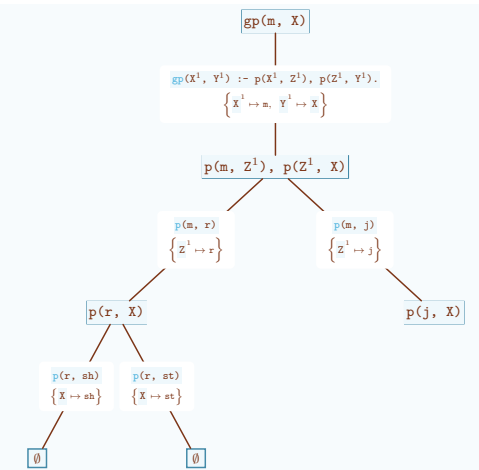
```
Fonction résolution(P, q_1 ∧ ··· ∧ q_n)
    Si n = 0
        Renvoyer {ι}
    Sinon
        Soit res := ∅
        Pour chaque clause t := t1, ..., tk de P
            Soit t' :- t1', ..., tk' := rafraichir(t :- t1, ..., tk)
            Soit σ := unifier(q_1, t')
            Soient t1'' := t1'·σ,  ...,  tk'' := tk'·σ
            Soit res' := résolution(P, t1'' ∧ ··· ∧ tk''  ∧ q_2 ∧ ··· ∧ q_n)
            Pour chaque σ' de res'
                Soit res := res ∪ {σ ∘ σ'}
            Fin
        Fin
        Renvoyer res
    Fin
Fin
```

La fonction `rafraichir` prend une clause en entrée et renomme toutes ses variables par des variables non encore utilisées.

Il est possible d'illustrer la résolution par un arbre, l'*arbre de résolution*. Chaque nœud contient le but courant et chaque arête est décorée par la clause sélectionnée et la substitution associée.

Exemple

Depuis l'exemple précédent, nous avons l'arbre de résolution



Les atomes ont été abrégés de sorte que

- `gp` signifie `grand_parent` ;
- `p` signifie `parent` ;
- `m` signifie `marvin` ;
- `r` signifie `randy` ;
- `j` signifie `jimbo` ;
- `sh` signifie `shelly` ;
- `st` signifie `stan`.

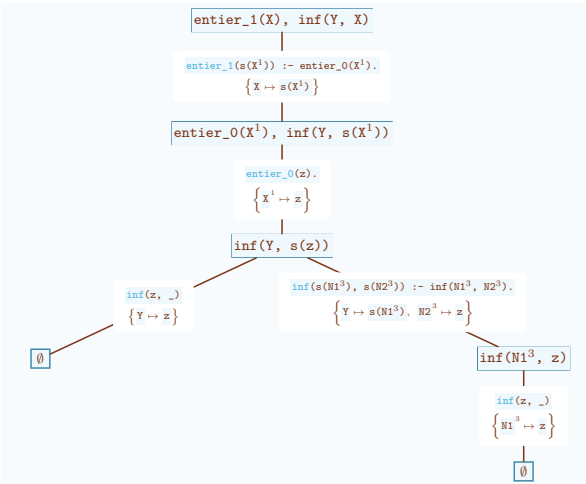
Reprenons les entiers de Peano et plus spécifiquement les deux prédicats et la requête suivants :

```
entier_0(z).
entier_1(s(X)) :-
    entier_0(X).
```

```
inf(z, _).
inf(s(N1), s(N2)) :-
    inf(N1, N2).
```

```
?- entier_1(X), inf(Y, X).
X = s(z), Y = z ;
X = Y, Y = s(z) ;
false.
```

L'arbre de résolution de cette requête est



Il contient deux solutions :

- $\{Y \mapsto z, X \mapsto s(z)\}$;
- $\{Y \mapsto s(z), X \mapsto s(z)\}$.

Elles s'obtiennent en composant les substitutions du bas vers le haut qui aboutissent aux succès $\emptyset$.

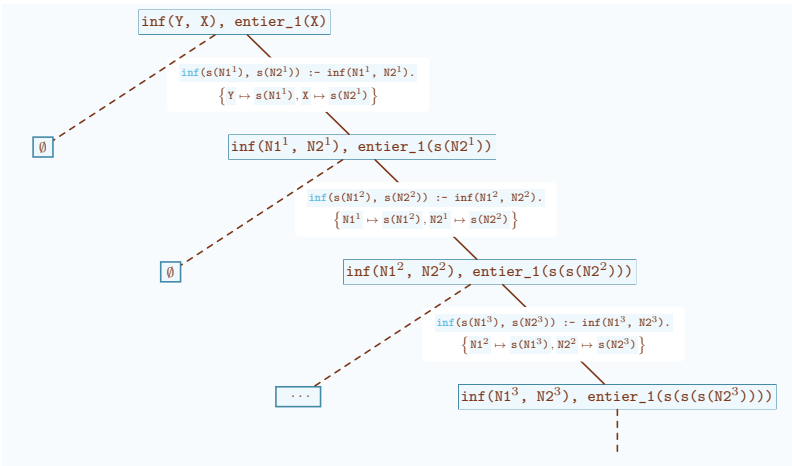
L'interpréteur imprime `false.` après la dernière solution car le but `inf(N1³, z)` est développé en sélectionnant la règle `inf(s(N1⁴), s(N2⁴)) :- inf(N1⁴, N2⁴).` de manière infructueuse.

La requête obtenue en changeant l'ordre de ses sous-buts donne

```
?- inf(Y, X), entier_1(X).
Y = z, X = s(z) ;
Y = X, X = s(z) ;
```

De manière surprenante, la résolution semble continuer après le calcul de ces deux solutions.

Ceci s'explique par l'arbre de résolution de cette requête qui est de la forme



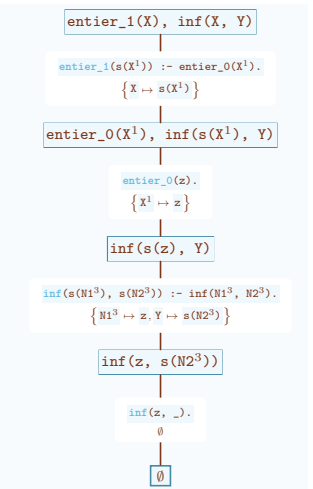
L'arbre de résolution possède une **branche infinie**.
Les deux solutions calculées correspondent aux deux buts $\emptyset$ rencontrés en considérant la clause `inf(z, _).` bifurquant de la branche infinie en son 1er et 2e nœud.
Il y a de plus, pour chaque autre nœud de la branche principale, une sélection de la clause `inf(z, _).` qui ne produit néanmoins aucune solution.
Point important : mettre en premier les parties les plus contraignantes dans les corps des clauses.

La requête obtenue en changeant l'ordre des variables dans `inf` donne

```
?- entier_1(X), inf(X, Y).
X = s(z), Y = s(_).
```

De manière surprenante, la résolution se termine après le calcul de cette solution.

Ceci s'explique par l'arbre de résolution de cette requête qui est



L'arbre de résolution est bien **fini**.

La solution calculée est $\{X \mapsto z, Y \mapsto s(N2^3)\}$. Comme $N2^3$ peut prendre n'importe quelle valeur (elle ne dépend ni de `X` ni de `Y`), SWI-Prolog l'affiche sous la forme d'un **joker** `_`.

Cette solution est finalement très cohérente : la requête initiale demande les `Y` qui sont supérieurs à `s(z)`. La réponse est l'ensemble des termes de la forme `s(_)`. Chacun de ces termes respecte la contrainte spécifiée.

Reprenons les prédicats `entier/1` et `add/3` la requête suivants :

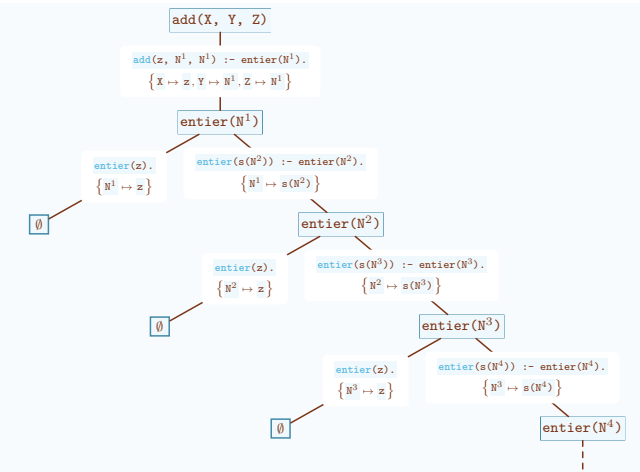
```
entier(z).
entier(s(N)) :- entier(N).
```

```
add(z, N, N) :-
    entier(N).
add(s(N1), N2, s(N3)) :-
    add(N1, N2, N3).
```

```
?- add(X, Y, Z).
X = Y, Y = Z, Z = z ;
X = z, Y = Z, Z = s(z) ;
X = z, Y = Z, Z = s(s(z))
```

Il y a une infinité de solutions.

Ceci s'explique par l'arbre de résolution de cette requête qui est de la forme



Il y a une **branche infinie** apportant une **infinité de solutions**.

À sa racine, la clause `add(s(N1), N2, s(N3)) :- add(N1, N2, N3).` n'est pas sélectionnée car la clause `add(z, N, N) :- entier(N).` produit un arbre infini.

Seules sont trouvées les solutions de la forme $\{X \mapsto z, Y \mapsto s(\ldots(z)\ldots), Z \mapsto s(\ldots(z)\ldots)\}$.

L'arbre de résolution est visité en **profondeur**. Un parcours en profondeur à profondeur bornée trouverait les autres solutions.

4.2.3. Négation

L'*hypothèse du monde clos* consiste à supposer que tout ce qui n'est pas connu comme vrai est considéré comme faux.

En programmation logique, cette hypothèse est adoptée. Ainsi, tout ce qui **n'est pas démontrable** à partir des prédicats du programme est **considéré comme faux**.

Exemple

Soit le programme

```
couleur(vert).
couleur(rouge).
```

et la requête

```
?- couleur(jaune).
false.
```

Cette requête est considérée comme **fausse** car elle est **indémontrable** par les clauses contenues dans le programme.

Attention : si une requête n'admet pas de solution, il est incorrect d'affirmer que sa négation admet une solution.

L'absence de démonstration d'une requête n'équivaut pas à l'existence d'une démonstration de sa négation.

L'**opérateur de négation par l'échec** est `\+`.

Exemple

Soient les trois prédicats

```
couleur(vert).
couleur(jaune).
couleur(rouge).
couleur(bleu).
```

```
claire(vert).
claire(jaune).
```

```
foncee(C) :-
    couleur(C),
    \+ claire(C).
```

Le prédicat `foncee/1` est tel que `foncee(C)` si `C` est une couleur telle que `claire(C)` échoue. Ainsi,

```
?- foncee(C).
C = rouge ;
C = bleu ;
false.
```

Exercice : développer l'arbre de résolution de la requête de cet exemple.

Il se construit de la même façon que les précédents, à la différence qu'un nœud de la forme `\+ q`, où `q` est un but, produit une solution si et seulement si `q` ne produit aucune solution.

L'**opérateur de différence** `\=` est la négation de l'unification. Ceci signifie que pour tous termes `t1` et `t2`, `t1 \= t2` si et seulement si `\+ (t1 = t2)`. Nous avons donc `t1 \= t2` si et seulement si `t1` et `t2` ne sont pas unifiables.

La négation par l'échec (incluant `\+` et `\=`) ne doit être employée que sur des termes dont **toutes les variables sont spécialisées**.

Exemple

Considérons le programme formé par l'unique prédicat `p(a).`.
Considérons les trois requêtes

```
?- p(X).
X = a.
```

```
?- X = b, p(X).
false.
```

```
?- X = b, \+ p(X).
X = b.
```

```
?- \+ p(X), X = b.
false.
```

- La 1re est immédiate : `X = a` est la seule solution de `p(X)`.
- La 2e est immédiate aussi : cela revient à regarder `p(b)` qui ne peut pas être démontrée.
- La 3e regarde si `p(b)` ne peut pas être démontrée, ce qui est bien le cas.
- La 4e, obtenue en transposant les deux sous-buts de la requête précédente, est surprenante. Comme la requête commence par `\+ p(X)` et que la variable `X` n'est à ce stade **pas spécialisée**, et comme `p(X)` admet une solution $\{X \mapsto a\}$, sa négation `\+ p(X)` échoue. Ainsi, la requête toute entière échoue.

4.3. Nombres et listes

4.3.1. Listes

Prolog possède une **syntaxe dédiée** pour les **listes**.

Voici les principales constructions :

- la **liste vide** est noté `[]` ;
- la liste contenant les termes `t1`, ..., `tn` est notée `[t1, ..., tn]` ;
- si `t` est un terme et `lst` est une liste, alors `[t | lst]` est la liste dont la **tête** est `t` et la **queue** est `lst`.

  Cette notation s'étend de la façon suivante : si `t1`, ..., `tn` sont des termes et `lst` est une liste, alors `[t1, ..., tn | lst]` est la liste dont les éléments sont, dans l'ordre, `t1`, ..., `tn` puis ceux de `lst`.

Exemples

Voici quelques tentatives d'unification de listes :

```
?- [b, a, X] = [X, a, Y].
X = Y, Y = b.
```

```
?- [a, b, c] = [X | Y].
X = a, Y = [b, c].
```

```
?- [X, b, X] = [a, b, c].
false.
```

```
?- [a, a, X | [c, b]] = [Y | Z].
Y = a, Z = [a, X, c, b].
```

Soit `membre/2` le prédicat tel que `membre(X, LST)` si `X` est un élément de la liste `LST`.

Il se définit par

```
membre(X, [X | _]).
membre(X, [_ | LST]) :-
    membre(X, LST).
```

Exemples

```
?- membre(a, [a, b, c]).
true ;
false.
```

```
?- membre(a, [b, a, c]).
true ;
false.
```

```
?- membre(a, [b, b, c]).
false.
```

```
?- membre(X, [b, b, c]).
X = b ;
X = b ;
X = c ;
false.
```

```
?- membre(X, [b, a, Y]).
X = b ;
X = a ;
X = Y ;
false.
```

```
?- membre(X, Y).
Y = [X|_] ;
Y = [_, X|_] ;
Y = [_, _, X|_] ;
Y = [_, _, _, X|_] ;
% ...
```

Il s'agit du prédicat `member/2` prédéfini.

Soit `tete/2` le prédicat tel que `tete(X, LST)` si `X` **apparaît en tête** de la liste non vide `LST`. Il se définit par

```
tete(X, [X | _]).
```

Soit `dernier/2` le prédicat tel que `dernier(X, LST)` si `X` **apparaît comme dernier élément** de la liste non vide `LST`. Il se définit par

```
dernier(X, [X]).
dernier(X, [_ | LST]) :-
    dernier(X, LST).
```

Exemples

```
?- tete(X, Y).
Y = [X|_].
```

```
?- dernier(X, [a, b, a, b]).
X = b ;
false.
```

```
?- dernier(a, L).
L = [a] ;
L = [_, a] ;
L = [_, _, a] ;
L = [_, _, _, a] ;
% ...
```

Soit `concatenation/3` le prédicat tel que `concatenation(LST_1, LST_2, LST_3)` si la liste `LST_3` est la concaténation des deux listes `LST_1` et `LST_2`. Il se définit par

```
concatenation([], LST, LST).
concatenation([X | LST_1], LST_2, [X | LST_3]) :-
    concatenation(LST_1, LST_2, LST_3).
```

Exemples

```
?- concatenation([a, b, c], [d, c], L).
L = [a, b, c, d, c].
```

```
?- concatenation([d, c], L, [d, c, b, b]).
L = [b, b].
```

```
?- concatenation(L1, [d, c], L2).
L1 = [], L2 = [d, c] ;
L1 = [_A], L2 = [_A, d, c] ;
L1 = [_A, _B], L2 = [_A, _B, d, c] ;
L1 = [_A, _B, _C], L2 = [_A, _B, _C, d, c] ;
% ...
```

```
?- concatenation(L, [d, c], [a, d, c]).
L = [a] .
```

```
?- concatenation([c], L, [d, c]).
false.
```

```
?- concatenation(L1, L2, [b, c, a, a]).
L1 = [], L2 = [b, c, a, a] ;
L1 = [b], L2 = [c, a, a] ;
L1 = [b, c], L2 = [a, a] ;
L1 = [b, c, a], L2 = [a] ;
L1 = [b, c, a, a], L2 = [] ;
false.
```

Il s'agit du prédicat `append/3` prédéfini.

Soit `prefixe/2` le prédicat tel que `prefixe(LST_1, LST_2)` si la liste `LST_1` est un préfixe de la liste `LST_2`. Il se définit par

```
prefixe(LST_1, LST_2) :-
    concatenation(LST_1, _, LST_2).
```

Soit `suffixe/2` le prédicat tel que `suffixe(LST_1, LST_2)` si la liste `LST_1` est un suffixe de la liste `LST_2`. Il se définit par

```
suffixe(LST_1, LST_2) :-
    concatenation(_, LST_1, LST_2).
```

Ils se basent sur l'idée que `LST_1` est un préfixe (resp. suffixe) de `LST_2` s'il existe une liste `LST_3` telle que `LST_2` est la concaténation de `LST_1` et `LST_3` (resp. `LST_3` et `LST_1`).

Exemples

```
?- prefixe(L, [a, b, c]).
L = [] ;
L = [a] ;
L = [a, b] ;
L = [a, b, c] ;
false.
```

```
?- suffixe(L, [a, b, c]).
L = [a, b, c] ;
L = [b, c] ;
L = [c] ;
L = [] ;
false.
```

Soit `facteur/2` le prédicat tel que `facteur(LST_1, LST_2)` si la liste `LST_1` **est un facteur** de la liste `LST_2`. Il se définit par

```
facteur(LST_1, LST_2) :-
    suffixe(LST_3, LST_2),
    prefixe(LST_1, LST_3).
```

Il se base sur l'idée que `LST_1` est un facteur de `LST_2` si `LST_1` est un préfixe d'un suffixe de `LST_2`.

Exemples

```
?- facteur([a, b], [c, a, b, b]).
true ;
false.
```

```
?- facteur([a, b], [c, a, c, b]).
false.
```

```
?- facteur(L, [a, b, b]).
L = [] ;
L = [a] ;
L = [a, b] ;
L = [a, b, b] ;
L = [] ;
```

```
L = [b] ;
L = [b, b] ;
L = [] ;
L = [b] ;
L = [] ;
false.
```

```
?- facteur([b, a], L).
L = [b, a|_] ;
L = [_, b, a|_] ;
L = [_, _, b, a|_] ;
L = [_, _, _, b, a|_] ;
L = [_, _, _, _, b, a|_] ;
% ...
```

L'idée précédente énonçant que `LST_1` est un facteur de `LST_2` si `LST_1` est un préfixe d'un suffixe de `LST_2` admet naturellement la **variante** suivante.

La liste `LST_1` est un facteur de `LST_2` si `LST_1` est un suffixe d'un prefixe de `LST_2`.

Cela nous permet de définir le prédicat `facteur2/2` tel que `facteur2(LST_1, LST_2)` si la liste `LST_1` **est un facteur** de la liste `LST_2` par

```
facteur2(LST_1, LST_2) :-
    prefixe(LST_1, LST_3),
    suffixe(LST_3, LST_2).
```

Cependant, ce prédicat n'est pas effectivement équivalent au précédent. Ceci est illustré par la requête ci-contre.

```
?- facteur2(L, [a, b]).
L = [] ;
L = [] ;
L = [] ;
L = [a] ;
L = [b] ;
L = [a, b] ;
% Résolution infinie.
```

Sa résolution s'engouffre dans une branche qui mène à un arbre de résolution infini, sans produire de nouvelles solutions.

La partie infinie de l'arbre commence au moment où un but `concatenate(L1, L2, L3)` est considéré.

Exercice : dessiner le début de l'arbre de résolution de la requête plus simple `facteur2(L, []).` qui produit déjà un arbre de résolution infini après avoir trouvé la solution `L = []`. Comparer avec la résolution de `facteur(L, []).`.

Soit `sous_liste/2` le prédicat tel que `sous_liste(LST_1, LST_2)` si la liste `LST_1` est une sous-liste de la liste `LST_2`. Il se définit par

```
sous_liste([], _).
sous_liste([X | LST_1], [X | LST_2]) :-
    sous_liste(LST_1, LST_2).
sous_liste(LST_1, [_ | LST_2]) :-
    sous_liste(LST_1, LST_2).
```

Exemples

```
?- sous_liste([a, c], [a, b, c]).
true ;
false.
```

```
?- sous_liste([c, a], [a, b, c]).
false.
```

```
?- sous_liste(L, [a, b, c]).
L = [] ;
L = [a] ;
L = [a, b] ;
L = [a, b, c] ;
L = [a, b] ;
L = [a] ;
L = [a, c] ;
L = [a] ;
```

```
L = [] ;
L = [b] ;
L = [b, c] ;
L = [b] ;
L = [] ;
L = [c] ;
L = [] ;
false.
```

```
?- sous_liste([a, b], L).
L = [a, b|_] ;
L = [a, b, _|_] ;
L = [a, b, _, _|_] ;
L = [a, b, _, _, _|_] ;
% ...
```

Soit `miroir/2` le prédicat tel que `miroir(LST_1, LST_2)` si la liste `LST_1` **est la liste miroir** de la liste `LST_2`. Il se définit par

```
miroir([], []).
miroir([X | LST_1], LST_2) :-
    miroir(LST_1, LST_3),
    concatenation(LST_3, [X], LST_2).
```

Il se base sur l'idée que la liste miroir d'une liste non vide peut s'exprimer en considérant l'image miroir de sa queue à laquelle est ajoutée en dernière position la tête de la liste originale.

Exemples

```
?- miroir([a, b, c], L).
L = [c, b, a].
```

```
?- miroir(L, [a, b, c]).
L = [c, b, a] ;
% Résolution infinie.
```

Exercice : dessiner le début de l'arbre de résolution de la requête `miroir(L, [a, b, c]).` pour observer comment l'arbre infini se développe après que la solution a été trouvée.

Considérons une version améliorée avec un paramètre jouant le rôle d'un accumulateur.

Voici un prédicat ternaire

```
miroir2_acc([], LST, LST).
miroir2_acc([X | LST_1], LST_2, LST_3) :-
    miroir2_acc(LST_1, [X | LST_2], LST_3).
```

et le prédicat binaire enrobant

```
miroir2(LST_1, LST_2) :-
    miroir2_acc(LST_1, [], LST_2).
```

Nous pouvons comparer les performances relatives des prédicats `miroir/2` par `miroir2/2` à l'aide du prédicat `time/1` qui est tel que `time(G)` résout le but `G` et imprime des informations sur sa résolution.

Exemples

```
?- time(miroir([a, b, c, d], L)).
 % 13 inferences, 0.000 CPU in 0.000 seconds
 (70% CPU, 622039 Lips)
L = [d, c, b, a].
```

```
?- time(miroir([a, b, c, d, e], L)).
 % 19 inferences, 0.000 CPU in 0.000 seconds
 (73% CPU, 978423 Lips)
L = [e, d, c, b, a].
```

```
?- time(miroir([a, b, c, d, e, f], L)).
 % 26 inferences, 0.000 CPU in 0.000 seconds
 (71% CPU, 1091749 Lips)
L = [f, e, d, c, b, a].
```

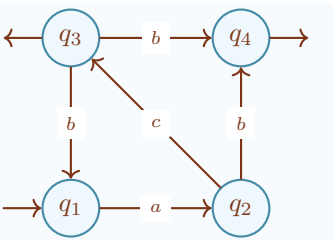
```
?- time(miroir2([a, b, c, d], L)).
 % 4 inferences, 0.000 CPU in 0.000 seconds
 (66% CPU, 211640 Lips)
L = [d, c, b, a].
```

```
?- time(miroir2([a, b, c, d, e], L)).
 % 5 inferences, 0.000 CPU in 0.000 seconds
 (74% CPU, 299724 Lips)
L = [e, d, c, b, a].
```

```
?- time(miroir2([a, b, c, d, e, f], L)).
 % 6 inferences, 0.000 CPU in 0.000 seconds
 (76% CPU, 263389 Lips)
L = [f, e, d, c, b, a].
```

Voici un autre exemple de manipulation de listes basé sur les mots reconnus par un **automate**.

Considérons l'automate



Le nœud (appelé `état`) q_1 est **initial**. Ceci est mentionné par la flèche entrante.

Les états q_3 et q_4 sont finaux. Ceci est mentionné par les flèches sortantes.

Les autres flèches (appelées `transitions`) sont décorées par des lettres.

Un mot u est *reconnu* par un automate s'il existe un chemin partant d'un état initial et arrivant dans un état final dont les décorations des transitions empruntées forment le mot u.

Par exemple, les mots ab, ac et $acbab$ sont reconnus par cet automate. À l'inverse, les mots a et abc ne sont pas reconnus par ce dernier.

Notre problématique est de représenter cet automate et de définir un prédicat permettant de vérifier (et donc, d'engendrer) les mots qu'il reconnaît.

Il faut tout d'abord représenter l'automate précédent.

Un automate étant caractérisé par l'information des états qu'il contient, de quels états sont initiaux ou terminaux et par ses transitions, nous fournissons les prédicats suivants :

```
etat(q1).
etat(q2).
etat(q3).
etat(q4).
```

```
initial(q1).

final(q3).
final(q4).
```

```
transition(q1, a, q2).
transition(q2, b, q4).
transition(q2, c, q3).
transition(q3, b, q1).
transition(q3, b, q4).
```

```
reconnu(MOT) :-
    initial(Q),
    reconnu(MOT, Q).
```

```
reconnu([], Q) :-
    final(Q).
reconnu([A | MOT], Q) :-
    etat(Q2),
    transition(Q, A, Q2),
    reconnu(MOT, Q2).
```

Notons que les prédicat `reconnu/1` et `reconnu/2`, du même nom mais ayant des arités différentes, sont différents.

Voici quelques requêtes :

```
?- reconnu([]).
false.
```

```
?- reconnu([a, b]).
true ;
false.
```

```
?- reconnu([a, c, b]).
true .
```

```
?- reconnu(MOT).
MOT = [a, c] ;
MOT = [a, c, b, a, c] ;
MOT = [a, c, b, a, c, b, a, c] ;
MOT = [a, c, b, a, c, b, a, c, b|...] ;
% ...
```

Voici quelques autres prédicats prédéfinis supplémentaires sur les listes.

- `append/2` tel que `append(LST_1, LST_2)` si l'aplatissement de la liste `LST_1` est la liste `LST_2`.

Exemple

```
?- append([[a], [b, c]], [a, b, c]).
true.
```

- `same_length/2` tel que `append(LST_1, LST_2)` si les listes `LST_1` et `LST_2` sont de la même longueur.

Exemple

```
?- same_length([a, b], [c, a]).
true.
```

- `select/3` tel que `select(X, LST_1, LST_2)` si la liste `LST_2` est obtenue en supprimant une occurrence de `X` dans la liste `LST_1`.

Exemple

```
?- select(a, [b, a, c, a], L).
L = [b, c, a] ;
L = [b, a, c].
```

Si `LST` est une liste, notons par `LST[`i`]` son i^{e} élément (commençant ici par convention à l'indice 1).

Pour tout $n \geqslant 1$, le prédicat prédéfini `maplist/`$n+1$ est tel que `maplist(P, LST_1, ..., LST_`n`)` si `P` est un prédicat d'arité n et `LST_1`, ..., `LST_`n sont des listes de longueur $k \geqslant 0$ telles que pour tout $1 \leqslant i \leqslant k$, `P(LST_1[`$i$`], ..., LST_`$n$`[`$i$`])`. Dans SWI-Prolog, ces prédicats sont définis pour tout $1 \leqslant n \leqslant 4$.

Exemples

En reprenant les entiers de Peano vu précédemment, nous pouvons écrire les requêtes suivantes.

- Pour $n = 2$:

```
?- maplist(fact, [s(z), z, s(s(s(z)))], L).
L = [s(z), s(z), s(s(s(s(s(z))))))].
```

- Pour $n = 3$:

```
?- maplist(add, [z, z, s(s(z))], [s(z), s(z), z], L).
L = [s(z), s(z), s(s(z))].
```

- Pour $n = 1$:

```
?- maplist(entier, [z, s(z), miaou]).
false.
```

Remarque : le prédicat `maplist/1` est tel que `maplist(P, LST)` si `P(X)` pour tout élément `X` de `LST`. Il permet donc d'effectuer un **test universel** (analogue au `List.for_all` d'OCaml).

4.3.2. Nombres et contraintes arithmétiques

Pour le moment, nous avons évité de travailler avec les **entiers natifs** de PROLOG. Ils existent bien mais sont limités dans la façon dont ils se comportent avec l'algorithme de résolution.

Nous allons utiliser ici la bibliothèque `clpfd` (*Constraint Logic Programming over Finite Domains*) pour manipuler les entiers et la plupart des opérations et relations sur ces derniers.

L'inclusion de cette bibliothèque se fait par

```
:- use_module(library(clpfd)).
```

Il ne faut pas oublier le `:-` lorsque l'inclusion s'effectue depuis un fichier `.pl`.

La *programmation par contrainte* est une branche de la programmation logique dans laquelle un problème est spécifié par des contraintes que doivent vérifier des objets comme des nombres ou des booléens.

Ici, nous allons considérer des problèmes sur les **entiers** soumis à des **contraintes arithmétiques**. La bibliothèque `clpfd` est parfaitement adaptée à ce contexte.

Une *expression arithmétique* est une expression pouvant prendre l'une des formes suivantes :

1. `C` ;
2. `X` ;
2. `-E` ;
3. `abs(E)` ;
4. `E1 + E2` ;
5. `E1 * E2` ;
6. `E1 - E2` ;
7. `E1 // E2` ;
8. `E1 ^ E2` ;
9. `min(E1, E2)` ;
10. `max(E1, E2)` ;
11. `mod(E1, E2)` ;

où `C` est une constante entière, `X` est une variable et `E`, `E1` et `E2` sont, récursivement, des expressions arithmétiques.

Leur sémantique est intuitive. En particulier, `^` est l'opération d'exponentiation et `//` est l'opération de division entière.

Exemple

```
1 // ((1 - X) ^ 2) + 1 // (1 - X * Y)
```

Une *contrainte arithmétique* est une expression pouvant l'une des formes suivantes :

1. `E1 #= E2` ;
2. `E1 #\= E2` ;
2. `E1 #>= E2` ;
3. `E1 #=< E2` ;
4. `E1 #> E2` ;
5. `E1 #< E2` ;

où `E1` et `E2` sont, récursivement, des expressions arithmétiques.

Leur sémantique est intuitive (attention néanmoins à la relation `#=<` notée de manière inhabituelle.

Exemple

```
X - Y #>= Z ^ 2 + 1
```

Nous pouvons utiliser l'approche par contrainte pour redéfinir de manière efficace les fonctions précédentes sur les entiers de Peano, en se servant des entiers manipulés via `clpfd`.

Exemple

```
fact(0, 1).
fact(N, F) :-
    N #> 0,
    N1 #= N - 1,
    F #= N * F1,
    fact(N1, F1).
```

```
?- fact(7, X).
X = 5040 ;
false.
```

```
?- fact(25, X).
X = 15511210043330985984000000 ;
false.
```

```
?- fact(X, 720).
X = 6 .
```

```
?- maplist(fact, [1, 2, 3, 4, 5, 6], L).
L = [1, 2, 6, 24, 120, 720] ;
false.
```

```
?- fact(X, X).
X = 1 ;
X = 2 ;
false.
```

```
?- X #=< 1024, fact(X, X * 2).
X = 3 ;
false.
```

Observons quand dans la toute dernière requête, la variante `fact(X, X * 2), X #=< 1024.` mène à une résolution infinie. Il est important de disposer les **contraintes les plus contraignantes en 1er** de sorte à **restreindre le plus possible l'espace de recherche**.

Exercice : implanter le prédicat `fibo/2` de manière similaire en utilisant `clpfd` et expérimenter.

Le prédicat infixe `in/2` permet de spécifier un contrainte par un **intervalle d'entiers**. Plus précisément,

```
X in A..B
```

exprime le fait que la variable `X` doit être supérieure à l'entier `A` et inférieure à l'entier `B`.

Exemples

```
?- X in 1..3, Y in 2..4.
X in 1..3, Y in 2..4.
```

```
?- X in 0..10, Y in 0..10, X #=< Y.
X in 0..10, Y#>=X, Y in 0..10.
```

Le prédicat `label/1` prend une liste de variables contraintes et **recherche explicitement leurs valeurs**.

Exemples

```
?- X in 1..2, Y in 2..3, label([X, Y]).
X = 1, Y = 2 ;
X = 1, Y = 3 ;
X = Y, Y = 2 ;
X = 2, Y = 3.
```

```
?- X in 0..2, Y in 0..2, X #=< Y, label([X, Y]).
X = Y, Y = 0 ;
X = 0, Y = 1 ;
X = 0, Y = 2 ;
X = Y, Y = 1 ;
X = 1, Y = 2 ;
X = Y, Y = 2.
```

Le prédicat infixe `ins/2` permet de spécifier un contrainte par un **intervalle d'entiers** sur toutes les **variables d'une liste**. Plus précisément,

```
LST ins A..B
```

exprime le fait que toutes les variables de la liste de variables `LST` doivent être supérieures à l'entier `A` et inférieures à l'entier `B`.

Le prédicat `all_distinct/1` permet de spécifier le fait que toutes les variables d'une liste doivent prendre des **valeurs distinctes**. Plus précisément,

```
all_distinct(LST)
```

exprime le fait que pour toutes variables `X1` et `X2` différentes, toute spécialisation de `X1` doit être différente de toute spécialisation de `X2`.

Exemple

```
?- [X, Y, Z] ins 0..1, all_distinct([X, Z]), label([X, Y, Z]).
X = Y, Y = 0, Z = 1 ;
X = 0, Y = Z, Z = 1 ;
X = 1, Y = Z, Z = 0 ;
X = Y, Y = 1, Z = 0.
```

Comme nous l'avons vu dans les nombreux exemples parcourus jusqu'à présent, si une requête admet plusieurs solutions, celles-ci sont calculées et imprimées une à une.

Il est cependant utile de pouvoir **calculer toutes les solutions directement**. Une bonne manière de faire consiste à construire une liste contenant toutes les substitutions vérifiant les contraintes. Pour cela, nous utilisons le prédicat `findall/3` tel que

```
findall(LST_VARS, C, RES)
```

si `LST_VARS` est une liste de variables, `C` une contrainte mettant en jeu des variables de `LST_VARS` et `RES` est la liste dont ses éléments sont des listes assignant des valeurs aux variables de `LST_VARS` qui rendent vraie la contrainte `C`.

Ceci est applicable dans le contexte des entiers mais pas uniquement.

Exemples

```
?- findall([X, Y], ([X, Y] ins -3..3, X + Y #= 0, label([X, Y])), R).
R = [[-3, 3], [-2, 2], [-1, 1], [0, 0], [1, -1], [2, -2], [3, -3]].
```

En reprenant l'exemple généalogique :

```
?- findall([X, Y], grand_parent(X, Y), R).
R = [[marvin, shelly], [marvin, stan], [grandma, stan], [grandma, shelly]].
```

Dans certains cas, suivant comment un prédicat est écrit et la façon dont l'algorithme de résolution travaille, plusieurs solutions identiques peuvent être collectées par `findall/3`.

Exemple

```
?- findall([X, Y], fratrie(X, Y), R).
R = [[randy, jimbo], [jimbo, randy], [shelly, stan], [stan, shelly], [stan, shelly], [shelly, stan]].
```

Il devient ainsi utile de **supprimer les doublons** dans une liste. Pour cela, nous utilisons le prédicat `sort/2` tel que `sort(LST_1, LST_2)` si la liste `LST_2` est la version triée et sans doublon de la liste `LST_1`.

Exemple

```
?- findall([X, Y], fratrie(X, Y), R1), sort(R1, R).
R1 = [[randy, jimbo], [jimbo, randy], [shelly, stan], [stan, shelly], [stan, shelly], [shelly, stan]],
R = [[jimbo, randy], [randy, jimbo], [shelly, stan], [stan, shelly]].
```

4.3.3. Exemples

Commençons par le cryptarithme

```
  S E N D
+ M O R E
----------
M O N E Y
```

qui consiste à rendre le calcul valide où chaque lettre désigne un chiffre différent.

Nous pouvons proposer le prédicat suivant pour le résoudre :

```
puzzle(S, E, N, D, M, O, R, Y) :-
    Vars = [S, E, N, D, M, O, R, Y],
    Vars ins 0..9,
    all_distinct(Vars),
                  S * 1000 + E * 100 + N * 10 + D
                + M * 1000 + O * 100 + R * 10 + E
    #= M * 10000 + O * 1000 + N * 100 + E * 10 + Y,
    M #\= 0,
    S #\= 0,
    label(Vars).
```

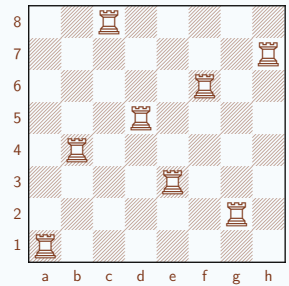
```
?- puzzle(S, E, N, D, M, O, R, Y).
S = 9, E = 5, N = 6, D = 7, M = 1, O = 0, R = 8, Y = 2 ;
false.
```

Il est possible d'utiliser la programmation par contrainte pour construire des nouveaux cryptarithmes avec des règles similaires (où une seule solution est possible). Mais cela est une autre histoire...

Exemple

Un *placement de tours* sur un échiquier de dimension n consiste à positionner n tours blanches de sorte qu'aucune n'en protège une autre.

L'exemple ci-contre montre une solution possible lorsque $n = 8$.



Nous cherchons un moyen de définir un prédicat qui calcule toutes les solutions étant donné n.

Comme d'habitude ici, le problème devient très simple une fois qu'il est bien modélisé.

Ici, nous pouvons remarquer qu'une solution est spécifiée entièrement par les indices des rangées de chacune des tours de la gauche vers la droite. Ceci est rendu possible par le fait qu'étant donnée une rangée, il y a exactement une tour qui l'occupe. La solution de l'exemple est ainsi représentée par la liste `[1, 4, 8, 5, 3, 6, 2, 7]`.

Nous avons ramené le problème initial à un problème spécifié par des contraintes. Une liste `R` est une réponse valide pour le problème des n tours si

- la longueur de `R` est n ;
- les éléments de `R` sont compris entre 1 et n ;
- les éléments de `R` sont tous distincts.

Ceci nous mène au prédicat `placement_tours/2` qui est tel que `placement_tours(N, R)` si la liste `R` est une solution au problème des `N` tours. Il est défini par

```
placement_tours(N, R) :-
    length(R, N),
    R ins 1..N,
    all_distinct(R),
    label(R).
```

```
?- placement_tours(3, R).
R = [1, 2, 3] ;
R = [1, 3, 2] ;
R = [2, 1, 3] ;
R = [2, 3, 1] ;
R = [3, 1, 2] ;
R = [3, 2, 1].
```

Voici un prédicat pour calculer toutes les solutions

```
placement_tours_toutes(N, SOL) :-
    findall([R], placement_tours(N, R), SOL).
```

```
?- placement_tours_toutes(2, R).
R = [[[1, 2]], [[2, 1]]].
```

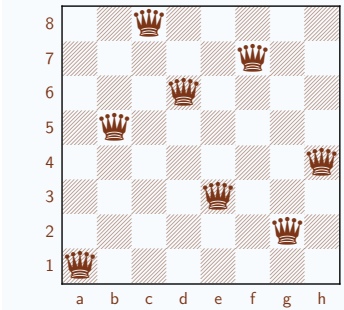
Exercice : changer un peu l'ordre des conditions définissant `placement_tours` et observer puis expliquer ce qu'il se passe.

Exercice : continuer ceci afin de dénombrer les solutions au problème en fonction de n.

Exemple

Un *placement de dames* sur un échiquier de dimension n consiste à positionner n dames noires de sorte qu'aucune n'en protège une autre.

L'exemple ci-contre montre une solution possible lorsque $n = 8$.



Nous cherchons un moyen de définir un prédicat qui calcule toutes les solutions étant donné n.

Tout comme pour le problème des n tours, nous allons coder une solution par la liste des rangées occupées par les dames, de la gauche vers la droite. Ainsi, la solution de l'exemple précédent est codée par la liste `[1, 5, 8, 6, 3, 7, 2, 4]`.

Soit le prédicat `independante/3` tel que `independante(LST, D, DIST)` si une dame placée sur la 1re colonne et en rangée `D` ne protège aucune des dames disposées à partir de la 2e colonne sur les rangées spécifiées par la liste `LST` avec `DIST = 1`. Il se définit par

```
independante([], _, _).
independante([D | DAMES], DAME, DIST) :-
    DAME #\= D,
    DAME #\= D + DIST,
    DAME #\= D - DIST,
    DIST_2 #= DIST + 1,
    independante(DAMES, DAME, DIST_2).
```

Exemples

```
?- independante([4, 2, 5], 1, 1).
true.
```

```
?- independante([4, 2, 5], 2, 1).
false.
% Protection orthogonale de la dame 2.
```

```
?- independante([4, 2, 5], 3, 1).
false.
% Protection diagonale de la dame 4.
```

```
?- independante([4, 2, 5], 4, 1).
false.
% Protection orthogonale de la dame 4.
```

```
?- independante([4, 2, 5], 5, 1).
false.
% Protection orthogonale de la dame 5.
```

```
?- independante([4, 2, 5], 6, 1).
true.
```

```
?- independante([4, 2, 5], 7, 1).
true.
```

```
?- independante([4, 2, 5], 8, 1).
false.
% Protection diagonale de la dame 5.
```

Soit le prédicat `independantes/1` tel que `independantes(DAMES)` si aucune dame de la liste de dames spécifiées par la liste `DAMES` n'en protège une autre. Il se définit par

```
independantes([]).
independantes([D | DAMES]) :-
    independante(DAMES, D, 1),
    independantes(DAMES).
```

Soit finalement le prédicat `placement_dames/2` tel que `placement_dames(N, R)` si la liste `R` est une solution au problème des `N` dames. Il est defini par

```
placement_dames(N, DAMES) :-
    length(DAMES, N),
    DAMES ins 1..N,
    all_distinct(DAMES),
    independantes(DAMES),
    label(DAMES).
```

Nous avons ainsi par exemple,

```
?- placement_dames(8, D).
D = [1, 5, 8, 6, 3, 7, 2, 4] ;
D = [1, 6, 8, 3, 7, 4, 2, 5] ;
D = [1, 7, 4, 6, 8, 2, 5, 3] ;
D = [1, 7, 5, 8, 2, 4, 6, 3] ;
% ...
```

```
?- findall([R], placement_dames(8, R), L), length(L, LEN).
L = [[[1, 5, 8, 6, 3, 7, 2|...]], [[1, 6, 8, 3, 7, 4|...]], % ...
LEN = 92.
```

INF6120 --- Programmation fonctionnelle et logique

Samuele Giraudo

Département d'informatique, Université du Québec à Montréal

giraudo.samuele@uqam.ca

Baccalauréat en informatique et génie logiciel

Automne 2025