Produit nommé

Étant donnés des types `T1`, ..., `Tn` et des identificateurs `ID1`, ..., `IDn`,

$$\{\texttt{ID1 : T1 ; ... ; IDn : Tn}\}$$

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

Produit nommé

Étant donnés des types `T1`, ..., `Tn` et des identificateurs `ID1`, ..., `IDn`,

$$\{\texttt{ID1 : T1 ; ... ; IDn : Tn}\}$$

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`.

Produit nommé

Étant donnés des types `T1`, ..., `Tn` et des identificateurs `ID1`, ..., `IDn`,

$$\{\texttt{ID1 : T1 ; ... ; IDn : Tn}\}$$

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`.

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

Produit nommé

Étant donnés des types `T1`, ..., `Tn` et des identificateurs `ID1`, ..., `IDn`,

$$\{\texttt{ID1 : T1 ; ... ; IDn : Tn}\}$$

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`.

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

Un enregistrement s'écrit

$$\{\texttt{ID1 = V1 ; ... ; IDn = Vn}\}$$

où `V1`, ..., `Vn` sont des valeurs de types respectifs `T1`, ..., `Tn`.

Produit nommé

Étant donnés des types `T1`, ..., `Tn` et des identificateurs `ID1`, ..., `IDn`,

$$\{\texttt{ID1 : T1 ; ... ; IDn : Tn}\}$$

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`.

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

Un enregistrement s'écrit

$$\{\texttt{ID1 = V1 ; ... ; IDn = Vn}\}$$

où `V1`, ..., `Vn` sont des valeurs de types respectifs `T1`, ..., `Tn`.

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

Accès aux champs d'un enregistrement

On accède au champ `c` d'un enregistrement `e` par

```
e.c
```

Accès aux champs d'un enregistrement

On accède au champ `c` d'un enregistrement `e` par

`e.c`

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

Accès aux champs d'un enregistrement

On accède au champ `c` d'un enregistrement `e` par

`e.c`

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

```
# 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>
```

« Modification » de champs d'un enregistrement

En programmation fonctionnelle, étant donné un nom `x`, il est **impossible de modifier la valeur** à laquelle `x` est lié.

« Modification » de champs d'un enregistrement

En programmation fonctionnelle, étant donné un nom `x`, il est **impossible de modifier la valeur** à laquelle `x` est lié.

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

« Modification » de champs d'un enregistrement

En programmation fonctionnelle, étant donné un nom `x`, il est **impossible de modifier la valeur** à laquelle `x` est lié.

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`.

« Modification » de champs d'un enregistrement

En programmation fonctionnelle, étant donné un nom `x`, il est **impossible de modifier la valeur** à laquelle `x` est lié.

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

$$\{\texttt{e with c1 = v1 ; ... ; cn = vn}\}$$

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`.

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

« Modification » de champs d'un enregistrement

En programmation fonctionnelle, étant donné un nom `x`, il est **impossible de modifier la valeur** à laquelle `x` est lié.

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

$$\{\texttt{e with c1 = v1 ; ... ; cn = vn}\}$$

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`.

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

« Modification » de champs d'un enregistrement

En programmation fonctionnelle, étant donné un nom `x`, il est **impossible de modifier la valeur** à laquelle `x` est lié.

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`.

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

« Modification » de champs d'un enregistrement

En programmation fonctionnelle, étant donné un nom `x`, il est **impossible de modifier la valeur** à laquelle `x` est lié.

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

$$\{\texttt{e with c1 = v1 ; ... ; cn = vn}\}$$

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`.

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

Contrainte sur les noms des champs

Considérons la situation suivante :

```
# type t1 = {a : int ; b : int};;
# type t2 = {a : int ; c : char};;
# let f x = x.a;;
```

Contrainte sur les noms des champs

Considérons la situation suivante :

```
# type t1 = {a : int ; b : int};;
# type t2 = {a : int ; c : char};;
# let f x = x.a;;
```

Problème : la fonction `f` ne peut pas être typée correctement. Le **champ** `a` est **commun** aux types `t1` et `t2` et ainsi, le type de l'expression `x.a` ne peut pas être déterminé.

Contrainte sur les noms des champs

Considérons la situation suivante :

```
# type t1 = {a : int ; b : int};;
# type t2 = {a : int ; c : char};;
# let f x = x.a;;
```

Problème : la fonction `f` ne peut pas être typée correctement. Le **champ** `a` est **commun** aux types `t1` et `t2` et ainsi, le type de l'expression `x.a` ne peut pas être déterminé.

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

Contrainte sur les noms des champs

Considérons la situation suivante :

```
# type t1 = {a : int ; b : int};;
# type t2 = {a : int ; c : char};;
# let f x = x.a;;
```

Problème : la fonction `f` ne peut pas être typée correctement. Le **champ** `a` est **commun** aux types `t1` et `t2` et ainsi, le type de l'expression `x.a` ne peut pas être déterminé.

L'interpréteur accepte tout de même ces commandes mais il néglige la définition du type `t1` (plus ancienne). Ainsi,

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

Contrainte sur les noms des champs

Considérons la situation suivante :

```
# type t1 = {a : int ; b : int};;
# type t2 = {a : int ; c : char};;
# let f x = x.a;;
```

Problème : la fonction `f` ne peut pas être typée correctement. Le **champ** `a` est **commun** aux types `t1` et `t2` et ainsi, le type de l'expression `x.a` ne peut pas être déterminé.

L'interpréteur accepte tout de même ces commandes mais il néglige la définition du type `t1` (plus ancienne). Ainsi,

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

Contrainte sur les noms des champs

Considérons la situation suivante :

```
# type t1 = {a : int ; b : int};;
# type t2 = {a : int ; c : char};;
# let f x = x.a;;
```

Problème : la fonction `f` ne peut pas être typée correctement. Le **champ** `a` est **commun** aux types `t1` et `t2` et ainsi, le type de l'expression `x.a` ne peut pas être déterminé.

L'interpréteur accepte tout de même ces commandes mais il néglige la définition du type `t1` (plus ancienne). Ainsi,

```
# 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>
```

Contrainte sur les noms des champs

Considérons la situation suivante :

```
# type t1 = {a : int ; b : int};;
# type t2 = {a : int ; c : char};;
# let f x = x.a;;
```

Problème : la fonction `f` ne peut pas être typée correctement. Le **champ** `a` est **commun** aux types `t1` et `t2` et ainsi, le type de l'expression `x.a` ne peut pas être déterminé.

L'interpréteur accepte tout de même ces commandes mais il néglige la définition du type `t1` (plus ancienne). Ainsi,

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

Contrainte sur les noms des champs

Considérons la situation suivante :

```
# type t1 = {a : int ; b : int};;
# type t2 = {a : int ; c : char};;
# let f x = x.a;;
```

Problème : la fonction `f` ne peut pas être typée correctement. Le **champ** `a` est **commun** aux types `t1` et `t2` et ainsi, le type de l'expression `x.a` ne peut pas être déterminé.

L'interpréteur accepte tout de même ces commandes mais il néglige la définition du type `t1` (plus ancienne). Ainsi,

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

Contrainte sur les noms des champs

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

Contrainte sur les noms des champs

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

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

Contrainte sur les noms des champs

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

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

On accède 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`.

Contrainte sur les noms des champs

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

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

On accède 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`.

Par exemple :

```
(* A.ml *)

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

Contrainte sur les noms des champs

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

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

On accède 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`.

Par exemple :

```
(* A.ml *)

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

```
(* B.ml *)

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

Contrainte sur les noms des champs

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

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

On accède 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`.

Par exemple :

```
(* A.ml *)

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

```
(* B.ml *)

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

```
(* C.ml *)

open A;;
open B;;

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

Plan

Somme

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

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

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

Somme

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

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

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

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

Somme

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

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

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

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

On appelle `Id1`, ..., `Idn` des constructeurs.

Somme

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

`Id1 | ... | Idn`

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

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

On appelle `Id1`, ..., `Idn` des constructeurs.

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

Somme

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

`Id1 | ... | Idn`

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

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

On appelle `Id1`, ..., `Idn` des constructeurs.

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

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

Somme

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

`Id1 | ... | Idn`

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

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

On appelle `Id1`, ..., `Idn` des constructeurs.

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

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

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

Somme

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

`Id1 | ... | Idn`

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

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

On appelle `Id1`, ..., `Idn` des constructeurs.

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

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

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

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

Somme

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

`Id1 | ... | Idn`

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

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

On appelle `Id1`, ..., `Idn` des constructeurs.

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

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

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

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

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

Somme

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

`Id1 | ... | Idn`

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

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

On appelle `Id1`, ..., `Idn` des constructeurs.

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

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

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

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

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

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

Constructeurs avec arguments

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

Constructeurs avec arguments

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`.

Constructeurs avec arguments

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`.

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

Constructeurs avec arguments

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`.

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

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

Constructeurs avec arguments

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`.

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

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

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

Constructeurs avec arguments

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`.

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

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

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

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

Constructeurs avec arguments

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`.

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

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

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

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

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

Exemple 1 — listes d'entiers

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

Exemple 1 — listes d'entiers

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** suivant :

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

Exemple 1 — listes d'entiers

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** suivant :

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

On construit des listes d'entiers de la manière suivante :

```
# let e1 = (Cellule (1, Vide));;
val e1 : liste_int = Cellule (1, Vide)
```

Exemple 1 — listes d'entiers

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** suivant :

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

On construit des listes d'entiers de la manière suivante :

```
# let e1 = (Cellule (1, Vide));;
val e1 : liste_int = Cellule (1, Vide)

# let e2 = (Cellule (2, e1));;
val e2 : liste_int = Cellule (2, Cellule (1, Vide))
```

Exemple 1 — listes d'entiers

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** suivant :

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

On construit des listes d'entiers de la manière suivante :

```
# let e1 = (Cellule (1, Vide));;
val e1 : liste_int = Cellule (1, Vide)

# let e2 = (Cellule (2, e1));;
val e2 : liste_int = Cellule (2, Cellule (1, Vide))

# let e3 = (Cellule (3, e2));;
val e3 : liste_int = Cellule (3, Cellule (2, Cellule (1, Vide)))
```

Exemple 1 — listes d'entiers

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** suivant :

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

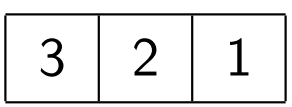
On construit des listes d'entiers de la manière suivante :

```
# let e1 = (Cellule (1, Vide));;
val e1 : liste_int = Cellule (1, Vide)

# let e2 = (Cellule (2, e1));;
val e2 : liste_int = Cellule (2, Cellule (1, Vide))

# let e3 = (Cellule (3, e2));;
val e3 : liste_int = Cellule (3, Cellule (2, Cellule (1, Vide)))
```

Le nom e3 est lié à la liste de valeur



Exemple 2 — arbres binaires d'entiers

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.

Exemple 2 — arbres binaires d'entiers

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** suivant :

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

Exemple 2 — arbres binaires d'entiers

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** suivant :

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

On construit des arbres binaires d'entiers de la manière suivante :

```
# let a1 = (Noeud (Vide, 1, Vide));;
val a1 : arbre_b_int = Noeud (Vide, 1, Vide)
```

Exemple 2 — arbres binaires d'entiers

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** suivant :

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

On construit des arbres binaires d'entiers de la manière suivante :

```
# let a1 = (Noeud (Vide, 1, Vide));;
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))
```

Exemple 2 — arbres binaires d'entiers

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** suivant :

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

On construit des arbres binaires d'entiers de la manière suivante :

```
# let a1 = (Noeud (Vide, 1, Vide));;
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));;
  Noeud (Noeud (Noeud (Vide, 1, Vide), 3, Noeud (Vide, 1, Vide)), 5,
   Noeud (Vide, 1, Vide))
```

Exemple 2 — arbres binaires d'entiers

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.

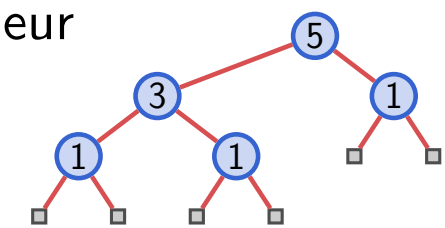
Cette définition se traduit en le type somme **récursif** suivant :

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

On construit des arbres binaires d'entiers de la manière suivante :

```
# let a1 = (Noeud (Vide, 1, Vide));;
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));;
  Noeud (Noeud (Noeud (Vide, 1, Vide), 3, Noeud (Vide, 1, Vide)), 5,
   Noeud (Vide, 1, Vide))
```

Le nom a3 est lié l'arbre binaire de valeur



Exemple 3 — arbres unaires binaires d'entiers

Un **arbre unaire binaire d'entiers** est

- l'arbre vide ou bien

Exemple 3 — arbres unaires binaires d'entiers

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

Exemple 3 — arbres unaires binaires d'entiers

Un **arbre unaire binaire d'entiers** est

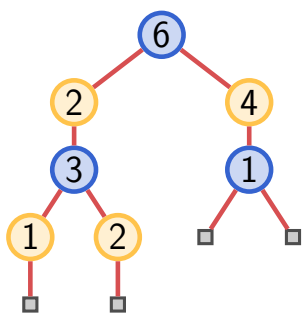
- l'arbre vide ou bien
- un nœud binaire contenant un entier, attaché à deux arbres unaires binaires dont les racines sont d'arités 1 ou bien
- un nœud unaire contenant un entier, attaché à un arbre unaire binaire dont la racine est d'arité 2.

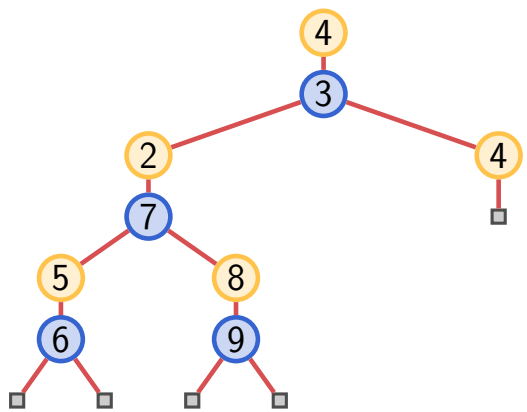
Exemple 3 — arbres unaires binaires d'entiers

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.

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





Exemple 3 — arbres unaires binaires d'entiers

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

Exemple 3 — arbres unaires binaires d'entiers

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

On commence 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;;
```

Exemple 3 — arbres unaires binaires d'entiers

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

On commence 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.

Exemple 3 — arbres unaires binaires d'entiers

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

On commence 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;;
```

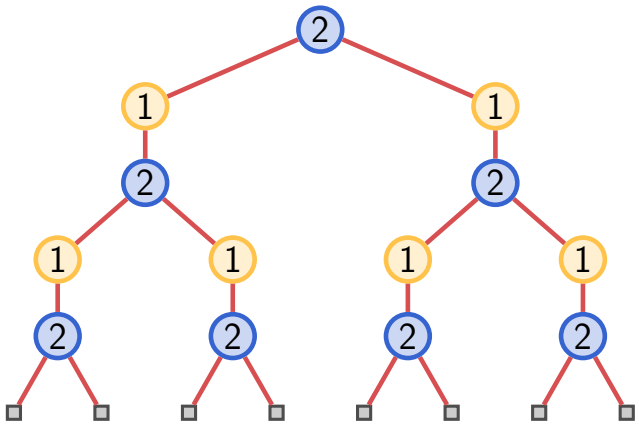
Le mot-clé `and` permet de réaliser des **définitions de types simultanées** : les types `arbre_1` et `arbre_2` sont en effet mutuellement récursifs.

On se base sur la définition des arbres unaires binaires pour construire finalement le type recherché :

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

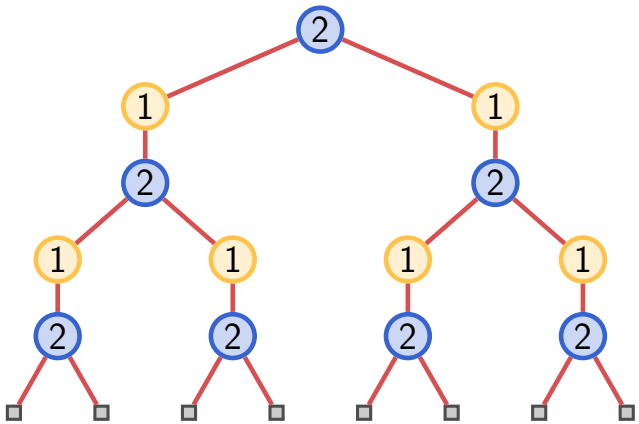
Exemple 3 — arbres unaires binaires d'entiers

Écrivons une expression de type `arbre_12` qui représente l'arbre ci-contre :



Exemple 3 — arbres unaires binaires d'entiers

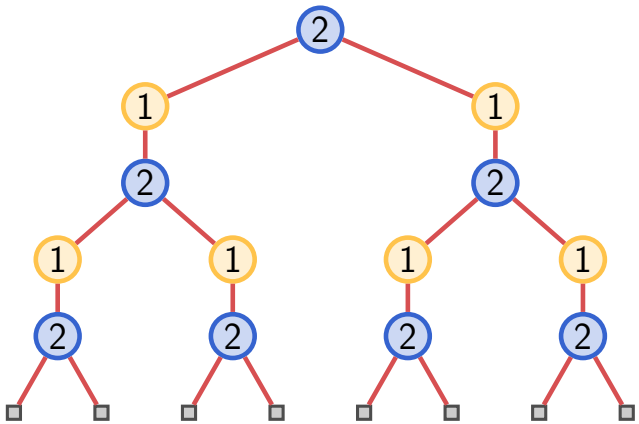
Écrivons une expression de type `arbre_12` qui représente l'arbre ci-contre :




```
# let a2 = (Noeud2 (Vide1, 2, Vide1)) in
```

Exemple 3 — arbres unaires binaires d'entiers

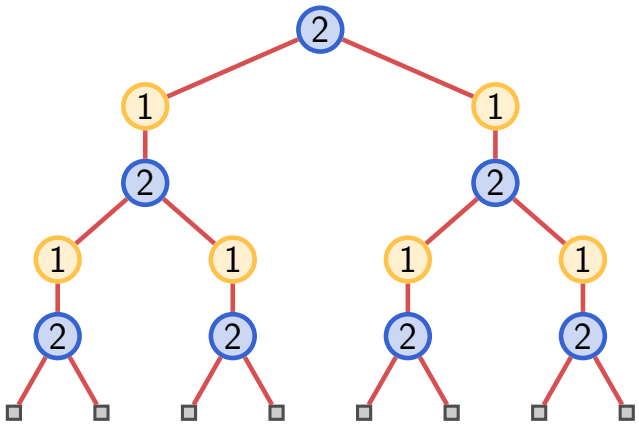
Écrivons une expression de type `arbre_12` qui représente l'arbre ci-contre :




```
# let a2 = (Noeud2 (Vide1, 2, Vide1)) in
      let a1 = (Noeud1 (1, a2)) in
```

Exemple 3 — arbres unaires binaires d'entiers

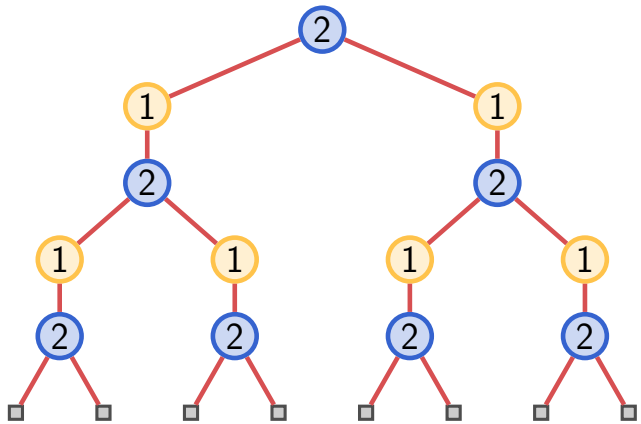
Écrivons une expression de type `arbre_12` qui représente l'arbre ci-contre :




```
# let a2 = (Noeud2 (Vide1, 2, Vide1)) in
      let a1 = (Noeud1 (1, a2)) in
          let a22 = (Noeud2 (a1, 2, a1)) in
```

Exemple 3 — arbres unaires binaires d'entiers

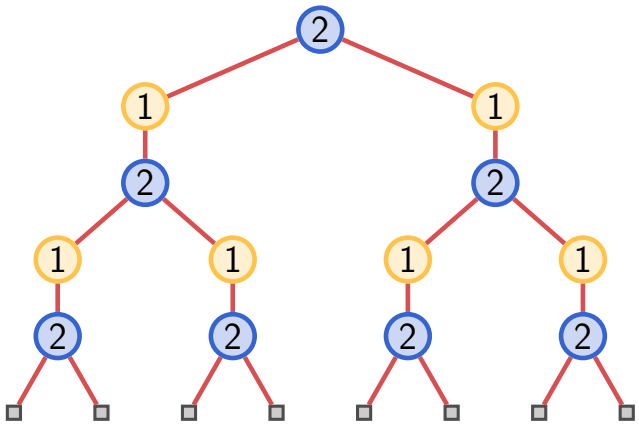
Écrivons une expression de type `arbre_12` qui représente l'arbre ci-contre :




```
# let a2 = (Noeud2 (Vide1, 2, Vide1)) in
      let a1 = (Noeud1 (1, a2)) in
          let a22 = (Noeud2 (a1, 2, a1)) in
              let a11 = (Noeud1 (1, a22)) in
```

Exemple 3 — arbres unaires binaires d'entiers

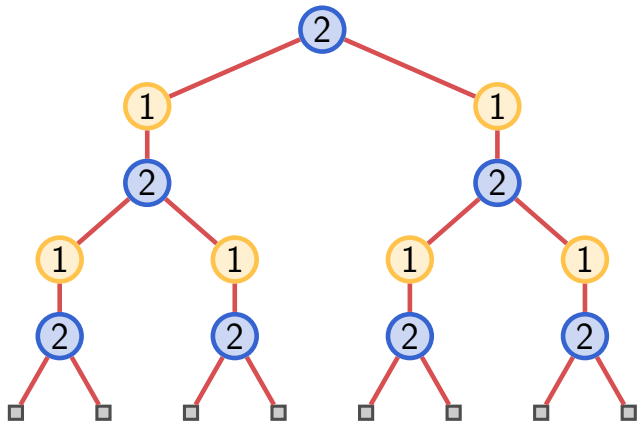
Écrivons une expression de type `arbre_12` qui représente l'arbre ci-contre :




```
# let a2 = (Noeud2 (Vide1, 2, Vide1)) in
      let a1 = (Noeud1 (1, a2)) in
          let a22 = (Noeud2 (a1, 2, a1)) in
              let a11 = (Noeud1 (1, a22)) in
                  (Arbre2 (Noeud2 (a11, 2, a11)));;
```

Exemple 3 — arbres unaires binaires d'entiers

Écrivons une expression de type `arbre_12` qui représente l'arbre ci-contre :




```
# let a2 = (Noeud2 (Vide1, 2, Vide1)) in
      let a1 = (Noeud1 (1, a2)) in
          let a22 = (Noeud2 (a1, 2, a1)) in
              let a11 = (Noeud1 (1, a22)) in
                  (Arbre2 (Noeud2 (a11, 2, a11)));;
- : 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))))))
```

Plan

Paramètres dans les types

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).

Paramètres dans les types

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).

On parle alors de types paramétrés.

Paramètres dans les types

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).

On parle alors 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 et `OP` fait intervenir des types, des opérateurs de types et `'P1`, ..., `'Pn`.

Paramètres dans les types

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).

On parle alors 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 et `OP` fait intervenir des types, des opérateurs de types et `'P1`, ..., `'Pn`.

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

Paramètres dans les types

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).

On parle alors 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 et `OP` fait intervenir des types, des opérateurs de types et `'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
```

Rôle des paramètres de type

Supposons que `T` soit un type défini par

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

Rôle des paramètres de type

Supposons que `T` soit un type défini par

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

On dit que `T` **est paramétré** par `'P1`, ..., `'Pn`.

Rôle des paramètres de type

Supposons que `T` soit un type défini par

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

On dit 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.

Rôle des paramètres de type

Supposons que `T` soit un type défini par

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

On dit 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.

Un type paramétré désigne ainsi un ensemble de types.

Rôle des paramètres de type

Supposons que `T` soit un type défini par

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

On dit 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.

Un type paramétré désigne ainsi un ensemble 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**) ;

Rôle des paramètres de type

Supposons que `T` soit un type défini par

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

On dit 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.

Un type paramétré désigne ainsi un ensemble 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**) ;

Rôle des paramètres de type

Supposons que `T` soit un type défini par

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

On dit 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.

Un type paramétré désigne ainsi un ensemble 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.

Exemple 1 — listes génériques

Le type à un paramètre

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

permet de représenter des listes dont chaque cellule contient une valeur d'un type quelconque e. On obtient ainsi des listes génériques.

Exemple 1 — listes génériques

Le type à un paramètre

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

permet de représenter des listes dont chaque cellule contient une valeur d'un type quelconque e. On obtient ainsi des listes génériques.

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

Exemple 1 — listes génériques

Le type à un paramètre

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

permet de représenter des listes dont chaque cellule contient une valeur d'un type quelconque e. On obtient ainsi des listes génériques.

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

Une liste de caractères :

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

Exemple 1 — listes génériques

Le type à un paramètre

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

permet de représenter des listes dont chaque cellule contient une valeur d'un type quelconque e. On obtient ainsi des listes génériques.

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

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)
```

Exemple 1 — listes génériques

Le type à un paramètre

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

permet de représenter des listes dont chaque cellule contient une valeur d'un type quelconque e. On obtient ainsi des listes génériques.

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

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
```

Exemple 2 — produit nommé à plusieurs paramètres

Le type à deux paramètres

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

permet de représenter des couples dont les coordonnées sont de types quelconques (et potentiellement différents).

Exemple 2 — produit nommé à plusieurs paramètres

Le type à deux paramètres

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

permet de représenter des couples dont les coordonnées sont de types quelconques (et potentiellement différents).

Il ne faut pas confondre avec le type

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

dans lequel les deux coordonnées doivent être d'un même type.

Exemple 2 — produit nommé à plusieurs paramètres

Le type à deux paramètres

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

permet de représenter des couples dont les coordonnées sont de types quelconques (et potentiellement différents).

Il ne faut pas confondre avec le type

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

dans lequel les deux coordonnées doivent être d'un même type.

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

Exemple 2 — produit nommé à plusieurs paramètres

Le type à deux paramètres

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

permet de représenter des couples dont les coordonnées sont de types quelconques (et potentiellement différents).

Il ne faut pas confondre avec le type

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

dans lequel les deux coordonnées doivent être d'un même type.

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

```
# let c = {x = 4 ; y = 'd'};;
val c : (int, char) couple = {x = 4; y = 'd'}
```

Exemple 2 — produit nommé à plusieurs paramètres

Le type à deux paramètres

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

permet de représenter des couples dont les coordonnées sont de types quelconques (et potentiellement différents).

Il ne faut pas confondre avec le type

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

dans lequel les deux coordonnées doivent être d'un même type.

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

```
# let c = {x = 4 ; y = 'd'};;
val c : (int, char) couple = {x = 4; y = 'd'}

# {c with y = 'e'};;
- : (int, char) couple = {x = 4; y = 'e'}
```

Exemple 2 — produit nommé à plusieurs paramètres

Le type à deux paramètres

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

permet de représenter des couples dont les coordonnées sont de types quelconques (et potentiellement différents).

Il ne faut pas confondre avec le type

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

dans lequel les deux coordonnées doivent être d'un même type.

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

```
# let c = {x = 4 ; y = 'd'};;
val c : (int, char) couple = {x = 4; y = 'd'}

# {c with y = 'e'};;
- : (int, char) couple = {x = 4; y = 'e'}

# {c with y = 2.2};;
- : (int, float) couple = {x = 4; y = 2.2}
```

Exemple 3 — arbres unaires binaires

On souhaite définir un type pour représenter les arbres unaires binaires où

- les nœud unaires contiennent tous des valeurs d'un même type ;

Exemple 3 — arbres unaires binaires

On souhaite 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.

Exemple 3 — arbres unaires binaires

On souhaite 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.

Exemple 3 — arbres unaires binaires

On souhaite 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
```

Exemple 3 — arbres unaires binaires

On souhaite 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;;
```

Exemple 3 — arbres unaires binaires

On souhaite 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;;
```

Exemple 4 — Mots

On cherche à représenter des mots sur un alphabet quelconque.

Exemple 4 — Mots

On cherche à représenter des mots sur un alphabet quelconque.

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

Exemple 4 — Mots

On cherche 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.

Par exemple, le mot $u := \texttt{baacba}$ vérifie $u_1 = \texttt{b}$ et $u_4 = \texttt{c}$.

Exemple 4 — Mots

On cherche 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.

Par exemple, le mot $u := \texttt{baacba}$ vérifie $u_1 = \texttt{b}$ 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.

Exemple 4 — Mots

On cherche 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.

Par exemple, le mot $u := \texttt{baacba}$ vérifie $u_1 = \texttt{b}$ 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 de faire mieux.

Exemple 4 — Mots

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

Exemple 4 — Mots

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 envoie chaque position i sur une lettre.

Exemple 4 — Mots

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 envoie chaque position i sur une lettre.

On obtient donc le type à un paramètre suivant

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

Exemple 4 — Mots

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 envoie chaque position i sur une lettre.

On obtient donc le type à un paramètre suivant

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

Quelques explications :

- le paramètre de type `'a` renseigne sur le type des lettres du mot ;

Exemple 4 — Mots

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 envoie chaque position i sur une lettre.

On obtient donc le type à un paramètre suivant

```
# 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 ;

Exemple 4 — Mots

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 envoie chaque position i sur une lettre.

On obtient donc le type à un paramètre suivant

```
# 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`).

Exemple 4 — Mots

```
# 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 `baabb`.

Exemple 4 — Mots

```
# 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 `baabb`.

```
# 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 `FTFTF...` de longueur 4294967296 (où `F` désigne `false` et `T` désigne `true`).

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

Exemple 5 — Images

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

Exemple 5 — Images

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.

Exemple 5 — Images

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 envoie chaque point sur une couleur. Ceci permet de représenter des images de très hautes définitions avec peu de mémoire.

Exemple 5 — Images

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 envoie chaque point sur une couleur. Ceci permet de représenter des images de très hautes définitions avec peu de mémoire.

On obtient les types

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

Exemple 5 — Images

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 envoie chaque point sur une couleur. Ceci permet de représenter des images de très hautes définitions avec peu de mémoire.

On obtient les types

```
# type point = int * int

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

Exemple 5 — Images

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 5 — Images

Avec de plus la définition du type

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

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

Par 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 `16x16` dont les pixels sur la diagonale sont gris et les autres noirs.

Exemple 5 — Images

Voici un autre 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 résolution) d'un disque noir sur un fond blanc.

Axe 3 : concepts avancés

Plan

Plan

Définitions récursives

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

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

Définitions récursives

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

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

Pour pouvoir réaliser des définitions récursives (c.-à-d. lier des valeurs à un nom en faisant référence au nom lui-même), on utilise la construction

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

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

Définitions récursives

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

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

Pour pouvoir réaliser des définitions récursives (c.-à-d. lier des valeurs à un nom en faisant référence au nom lui-même), on utilise la construction

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

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

```
# 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
```

Définitions récursives

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.

Définitions récursives

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.

Comparons les expressions suivantes :

```
# let x = 10 in
      let x = 20 in
          x + x;;
```

```
# let rec x =
      let x = 20 in
          x + x;;
```

```
# let x = 10 in
      (let x = 20 in
          x)
      + x;;
```

```
# let rec x =
      (let x = 20 in
      x)
      + x;;
```

Définitions récursives

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.

Comparons les expressions suivantes :

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

```
# let rec x =
      let x = 20 in
          x + x;;
```

```
# let x = 10 in
      (let x = 20 in
          x)
      + x;;
```

```
# let rec x =
      (let x = 20 in
      x)
      + x;;
```

Définitions récursives

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.

Comparons les expressions suivantes :

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

```
# let rec x =
      let x = 20 in
          x + x;;
```

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

```
# let rec x =
      (let x = 20 in
      x)
      + x;;
```

Définitions récursives

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.

Comparons les expressions suivantes :

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

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

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

```
# let rec x =
      (let x = 20 in
      x)
      + x;;
```

Définitions récursives

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.

Comparons les expressions suivantes :

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

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

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

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

Définitions récursives

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

```
# let f x =
      let rec f y =
          y + (f (x - 1))
      in
      if x = 0 then
          0
      else
          x + (f (x - 1));;
```

```
# let rec f x =
      let f y =
          y + (f (x - 1))
      in
      if x = 0 then
          0
      else
          x + (f (x - 1));;
```

```
# let rec f x =
      let rec f y =
          y + (f (x - 1))
      in
      if x = 0 then
          0
      else
          x + (f (x - 1));;
```

Définitions récursives

```
# 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));;
```

```
# let rec f x =
      let f y =
          y + (f (x - 1))
      in
      if x = 0 then
          0
      else
          x + (f (x - 1));;
```

```
# let rec f x =
      let rec f y =
          y + (f (x - 1))
      in
      if x = 0 then
          0
      else
          x + (f (x - 1));;
```

Définitions récursives

```
# 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));;
```

```
# 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>
```

```
# let rec f x =
      let rec f y =
          y + (f (x - 1))
      in
      if x = 0 then
          0
      else
          x + (f (x - 1));;
```

Définitions récursives

```
# 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));;
```

```
# 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);;
```

```
# let rec f x =
      let rec f y =
          y + (f (x - 1))
      in
      if x = 0 then
          0
      else
          x + (f (x - 1));;
```

Définitions récursives

```
# 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));;
```

```
# 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));;
```

Définitions récursives

```
# 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>
```

```
# 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));;
```

Définitions récursives

```
# 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);;
```

```
# 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));;
```

Définitions récursives

```
# 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 during evaluation
(looping recursion?).
```

```
# 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));;
```

Définitions récursives

```
# 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 during evaluation
(looping recursion?).
```

```
# 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>
```

Définitions récursives

```
# 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 during evaluation
(looping recursion?).
```

```
# 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);;
```

Définitions récursives

```
# 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 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 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 during evaluation
(looping recursion?).
```

```
# 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 during evaluation
(looping recursion?).
```

Définitions mutuellement récursives

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

Il est ainsi possible de définir des fonctions mutuellement récursives.

Définitions mutuellement récursives

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

Il est ainsi possible de définir des fonctions mutuellement récursives.

```
# 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.

Définitions mutuellement récursives

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

Il est ainsi possible de définir des fonctions mutuellement récursives.

```
# 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`.

Définitions mutuellement récursives

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

Il est ainsi possible de définir des fonctions mutuellement récursives.

```
# 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)` :

Définitions mutuellement récursives

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

Il est ainsi possible de définir des fonctions mutuellement récursives.

```
# 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)`

Définitions mutuellement récursives

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

Il est ainsi possible de définir des fonctions mutuellement récursives.

```
# 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)` $\rightarrow$ `(un 3)`

Définitions mutuellement récursives

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

Il est ainsi possible de définir des fonctions mutuellement récursives.

```
# 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)` $\rightarrow$ `(un 3)` $\rightarrow$ `(deux 2)`

Définitions mutuellement récursives

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

Il est ainsi possible de définir des fonctions mutuellement récursives.

```
# 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)` :

$$(\texttt{zero 4}) \to (\texttt{un 3}) \to (\texttt{deux 2}) \to (\texttt{zero 1})$$

Définitions mutuellement récursives

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

Il est ainsi possible de définir des fonctions mutuellement récursives.

```
# 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)` :

$$(\texttt{zero 4}) \to (\texttt{un 3}) \to (\texttt{deux 2}) \to (\texttt{zero 1}) \to (\texttt{un 0})$$

Définitions mutuellement récursives

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

Il est ainsi possible de définir des fonctions mutuellement récursives.

```
# 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)` :

$$\texttt{(zero 4)} \to \texttt{(un 3)} \to \texttt{(deux 2)} \to \texttt{(zero 1)} \to \texttt{(un 0)} \to \texttt{"un"}.$$