

Les types de données dans OCaml

CSI 3520

Amy Felty

University of Ottawa

OCaml jusqu'à présent

- On a vu un certain nombre de types de base :
 - int
 - float
 - char
 - string
 - bool
- On a vu quelques types structurés :
 - les paires
 - les tuples
 - les options
 - les listes
- On va maintenant voir des moyens plus généraux de définir de nouveaux types de données et de nouvelles structures de données.

Abréviations de type

- On a déjà vu quelques abréviations de type:

```
type point = float * float
```

Abréviations de type

- On a déjà vu quelques abréviations de type:

```
type point = float * float
```

- Les abréviations peuvent être une documentation utile

```
let distance (p1:point) (p2:point) : float =
  let square x = x *. x in
  let (x1,y1) = p1 in
  let (x2,y2) = p2 in
  sqrt (square (x2 -. x1) +. square (y2 -. y1))
```

- Mais les abréviations *n'ajoutent rien* au langage
 - elles sont *égales* à tous égards à un type existant

Abréviations de type

- On a déjà vu quelques abréviations de type:

```
type point = float * float
```

- On aurait pu écrire

```
let distance (p1:float*float)
              (p2:float*float) : float =
  let square x = x *. x in
  let (x1,y1) = p1 in
  let (x2,y2) = p2 in
  sqrt (square (x2 -. x1) +. square (y2 -. y1))
```

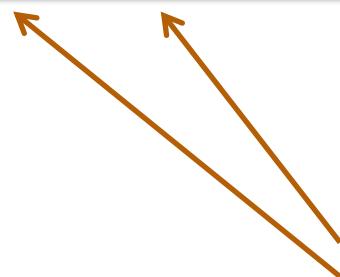
- Comme les types sont égaux, on peut *substituer* la définition du nom partout où on veut
 - on n'a pas ajouté de nouvelles structures de données

LES TYPES DE DONNÉES

Les types de données

- OCaml fournit un mécanisme général appelé **type de données** pour définir de nouvelles structures de données qui se composent de nombreuses alternatives

```
type my_bool = Tru | Fal
```



une **valeur** de type **my_bool** est l'une des deux:

- soit **Tru**
- soit **Fal**

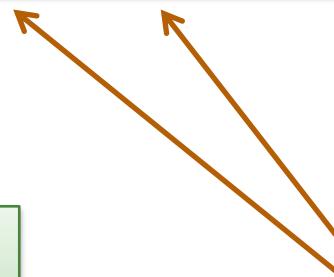
la barre verticale « | » est lue comme « ou »

Les types de données

- OCaml fournit un mécanisme général appelé **type de données** pour définir de nouvelles structures de données qui se composent de nombreuses alternatives

```
type my_bool = Tru | Fal
```

Tru et Fal sont appelés
« constructeurs »



une **valeur** de type **my_bool**
est l'une des deux:

- soit **Tru**
- soit **Fal**

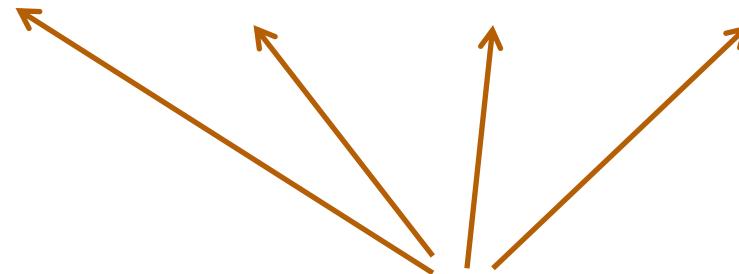
la barre verticale « | »
est lue comme « ou »

Les types de données

- OCaml fournit un mécanisme général appelé **type de données** pour définir de nouvelles structures de données qui se composent de nombreuses alternatives

```
type my_bool = Tru | Fal
```

```
type color = Blue | Yellow | Green | Red
```



il n'y a pas de restriction à
2 cas; on peut définir autant
d'alternatives que l'on veut

Les types de données

- OCaml fournit un mécanisme général appelé **type de données** pour définir de nouvelles structures de données qui se composent de nombreuses alternatives

```
type my_bool = Tru | Fal
```

```
type color = Blue | Yellow | Green | Red
```

- Créer des valeurs :

```
let b1 : my_bool = Tru  
let b2 : my_bool = Fal  
let c1 : color = Yellow  
let c2 : color = Red
```

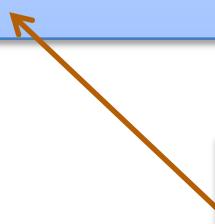
utilisez des constructeurs pour créer des valeurs

Les types de données

```
type color = Blue | Yellow | Green | Red  
  
let c1 : color = Yellow  
let c2 : color = Red
```

- en utilisant des valeurs de type de données :

```
let print_color (c:color) : unit =  
  match c with  
  | Blue ->  
  | Yellow ->  
  | Green ->  
  | Red ->
```



Utilisez le filtrage pour déterminer la couleur; puis implementez selon la tâche souhaitée

Les types de données

```
type color = Blue | Yellow | Green | Red

let c1 : color = Yellow
let c2 : color = Red
```

- en utilisant des valeurs de type de données :

```
let print_color (c:color) : unit =
  match c with
  | Blue -> print_string "blue"
  | Yellow -> print_string "yellow"
  | Green -> print_string "green"
  | Red -> print_string "red"
```

Les types de données

```
type color = Blue | Yellow | Green | Red

let c1 : color = Yellow
let c2 : color = Red
```

- en utilisant des valeurs de type de données :

```
let print_color (c:color) : unit =
  match c with
  | Blue -> print_string "blue"
  | Yellow -> print_string "yellow"
  | Green -> print_string "green"
  | Red -> print_string "red"
```

Pourquoi ne pas simplement utiliser des chaînes pour représenter les couleurs au lieu de définir un nouveau type?

Les types de données

```
type color = Blue | Yellow | Green | Red
```

erreur! :

```
let print_color (c:color) : unit =
  match c with
  | Blue -> print_string "blue"
  | Yellow -> print_string "yellow"
  | Red -> print_string "red"
```



Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Green

Les types de données

```
type color = Blue | Yellow | Green | Red
```

erreur! :

```
let print_color (c:color) : unit =
  match c with
  | Blue -> print_string "blue"
  | Yellow -> print_string "yellow"
  | Red -> print_string "red"
```



Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Green

Les définitions de types de données d'OCaml vous permettent de créer des types qui contiennent *précisément* les valeurs que vous voulez!

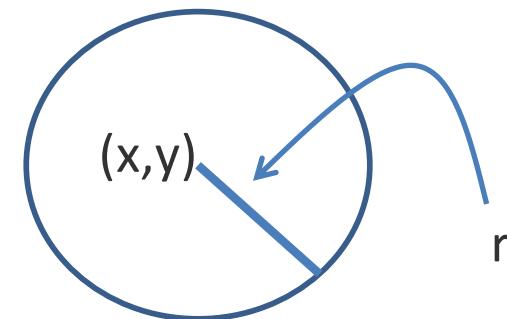
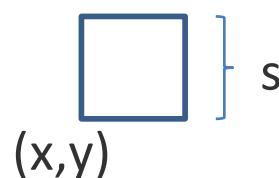
Les types de données peuvent avoir des arguments

- Les types de données sont plus que des énumérations de constantes :

```
type point = float * float

type simple_shape =
    Circle of point * float
  | Square of point * float
```

- On le lit comme suit: un **simple_shape** est:
 - soit un **cercle**, qui contient une **paire** d'un **point** et d'un **nombre à virgule flottante**,
 - soit un **carré** qui contient une **paire** d'un **point** et d'un **nombre à virgule flottante**



Les types de données peuvent avoir des arguments

- Les types de données sont plus que des énumérations de constantes :

```
type point = float * float
```

```
type simple_shape =
  Circle of point * float
| Square of point * float
```

```
let origin : point = (0.0, 0.0)
```

```
let circ1 : simple_shape = Circle (origin, 1.0)
```

```
let circ2 : simple_shape = Circle ((1.0, 1.0), 5.0)
```

```
let square : simple_shape = Square (origin, 2.3)
```

Les types de données peuvent avoir des arguments

- Les types de données sont plus que des énumérations de constantes :

```
type point = float * float

type simple_shape =
  Circle of point * float
| Square of point * float

let simple_area (s:simple_shape) : float =
  match s with
  | Circle (_, radius) -> 3.14 *. radius *. radius
  | Square (_, side) -> side *. side
```

Une comparaison

- Les types de données sont plus que des énumérations de constantes :

```
type point = float * float

type simple_shape =
  Circle of point * float
| Square of point * float

let simple_area (s:simple_shape) : float =
  match s with
  | Circle (_, radius) -> 3.14 *. radius *. radius
  | Square (_, side) -> side *. side
```

```
type my_shape = point * float

let simple_area (s:my_shape) : float =
  (3.14 *. radius *. radius) ?? or ?? (side *. side)
```

Des formes plus générales

```

type point = float * float

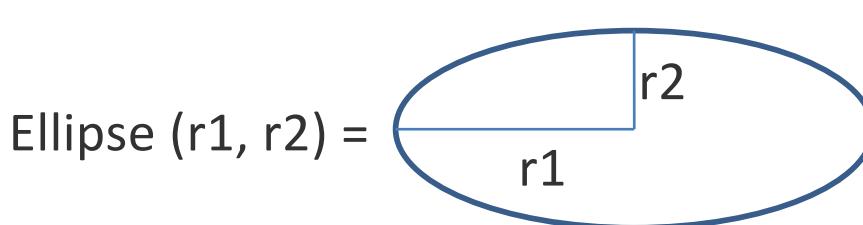
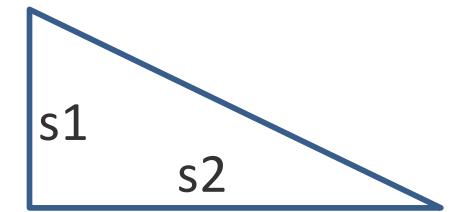
type shape =
  Square of float
  | Ellipse of float * float
  | RtTriangle of float * float
  | Polygon of point list

```

Square s =

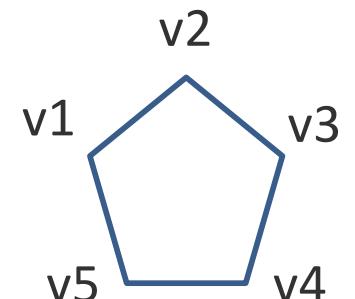


RtTriangle (s1, s2) =



Ellipse (r1, r2) =

Polygon [v1; ...; v5] =



Des formes plus générales

```
type point = float * float
```

```
type radius = float
```

```
type side = float
```

```
type shape =
```

 Square of side

 | Ellipse of radius * radius

 | RtTriangle of side * side

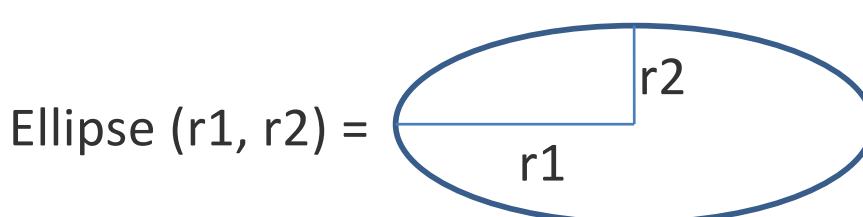
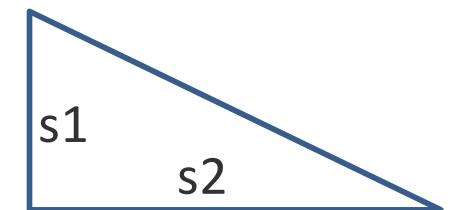
 | Polygon of point list

Les abréviations de type peuvent rendre la définition plus lisible

Square s =

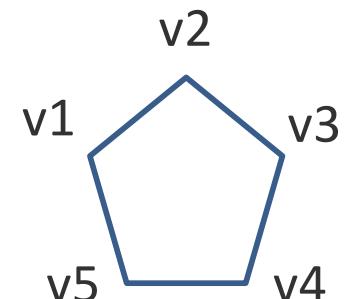


RtTriangle (s1, s2) =



Ellipse (r1, r2) =

RtTriangle [v1; ...; v5] =



Des formes plus générales

```
type point = float * float
type radius = float
type side = float
```

Square est construit
d'un seul côté

```
type shape =
  Square of side
  | Ellipse of radius * radius
  | RtTriangle of side * side
  | Polygon of point list
```

Ellipse est construit
à partir de deux rayons

```
let sq   : shape = Square 17.0
let ell  : shape = Ellipse (1.0, 2.0)
let rt   : shape = RtTriangle (1.0, 1.0)
let poly : shape = Polygon [(0., 0.); (1., 0.); (0.; 1.)]
```

RtTriangle est construit
à partir de deux côtés

ce sont toutes des formes;
ils sont construits de
différentes manières

Un polygone est construit à partir
d'une liste de points
(où chaque point est lui-même une paire)

Des formes plus générales

```
type point = float * float
type radius = float
type side = float

type shape =
  Square of side
  | Ellipse of radius * radius
  | RtTriangle of side * side
  | Polygon of point list
```

un type de données
définit également un
motif pour le filtrage

```
let area (s : shape) : float =
  match s with
  | Square s ->
  | Ellipse (r1, r2)->
  | RtTriangle (s1, s2) ->
  | Polygon ps ->
```

Des formes plus générales

```

type point = float * float
type radius = float
type side = float

type shape =
  Square of side
  | Ellipse of radius * radius
  | RtTriangle of side * side
  | Polygon of point list

```

un type de données définit également un motif pour le filtrage

```

let area (s : shape) : float =
  match s with
  | Square s ->
  | Ellipse (r1, r2) ->
  | RtTriangle (s1, s2) ->
  | Polygon ps ->

```

Square a un argument de type **float**, donc **s** est un motif pour les **valeurs** de type **float**

RtTriangle a un argument de type **float * float** donc **(s1, s2)** est un motif pour les valeurs de ce type

Des formes plus générales

```
type point = float * float
type radius = float
type side = float

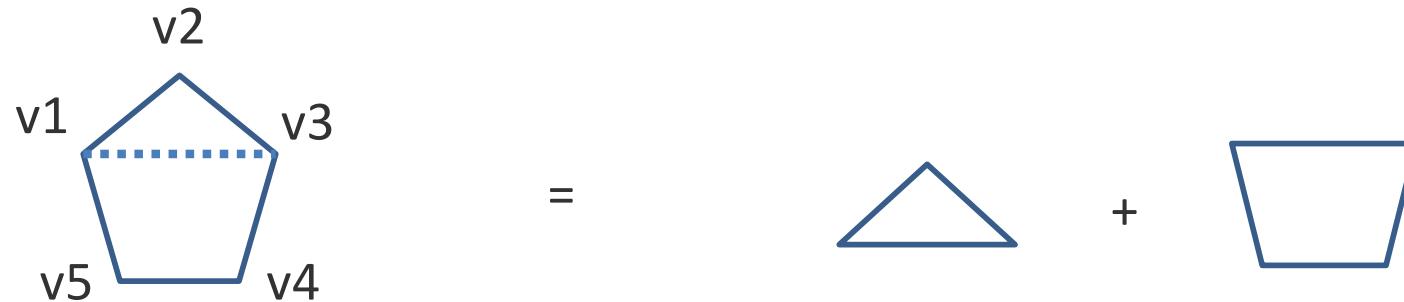
type shape =
  Square of side
  | Ellipse of radius * radius
  | RtTriangle of side * side
  | Polygon of point list
```

un type de données
définit également un
motif pour le filtrage

```
let area (s : shape) : float =
  match s with
  | Square s -> s *. s
  | Ellipse (r1, r2) -> pi *. r1 *. r2
  | RtTriangle (s1, s2) -> s1 *. s2 /. 2.
  | Polygon ps -> ???
```

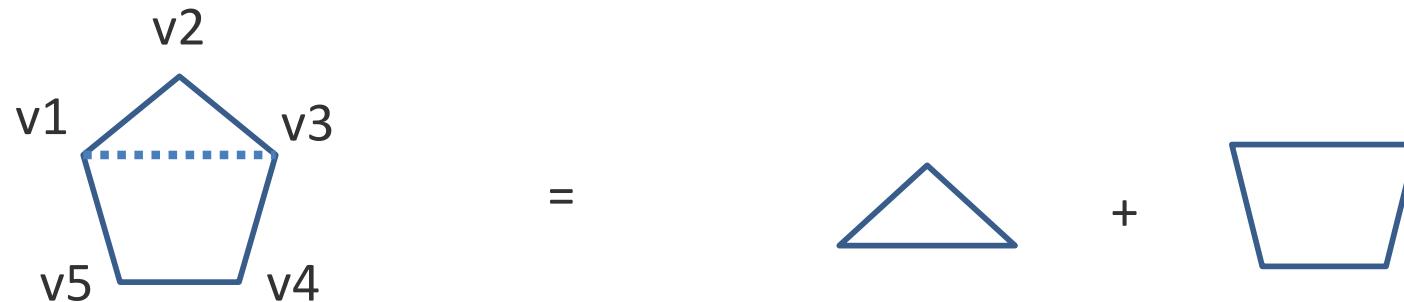
Calculer l'aire d'une forme

- Comment calculer la surface d'un polygone?
- Pour les polygones convexes :
 - Si le polygone a moins de 3 points :
 - La surface du polygone est 0! (C'est une ligne ou un point ou rien du tout.)
 - Si le polygone a plus de 3 points :
 - Calculez l'aire du triangle formé par les 3 premiers sommets.
 - Supprimez le deuxième sommet pour former un nouveau polygone
 - Calculer la somme de l'aire du triangle et du nouveau polygone



Calculer l'aire d'une forme

- Comment calculer la surface d'un polygone?
- Pour les polygones convexes :
 - Si le polygone a moins de 3 points :
 - La surface du polygone est 0! (C'est une ligne ou un point ou rien du tout.)
 - Si le polygone a plus de 3 points :
 - Calculez l'aire du triangle formé par les 3 premiers sommets.
 - Supprimez le deuxième sommet pour former un nouveau polygone
 - Calculer la somme de l'aire du triangle et du nouveau polygone
- Notez: Ceci est un bel **algorithme inductif**:
 - la surface d'un polygone à **n** points est calculée en termes de polygone plus petit à **n-1** points!

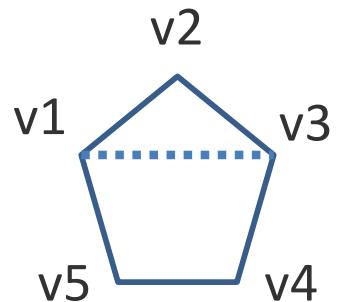


Calculer l'aire d'une forme

```
let area (s : shape) : float =
  match s with
  | Square s -> s *. s
  | Ellipse (r1, r2) -> r1 *. r2
  | RtTriangle (s1, s2) -> s1 *. s2 /. 2.
  | Polygon ps -> poly_area ps
```

Ce motif indique que la liste contient au moins 3 éléments

```
let rec poly_area (ps : point list) : float =
  match ps with
  | p1 :: p2 :: p3 :: tail ->
    tri_area p1 p2 p3 +. poly_area (p1 :: p3 :: tail)
  | _ -> 0.
```



=



Calculer l'aire d'une forme

```
let tri_area (p1:point) (p2:point) (p3:point) : float =
  let a = distance p1 p2 in
  let b = distance p2 p3 in
  let c = distance p3 p1 in
  let s = 0.5 *. (a +. b +. c) in
  sqrt (s *. (s -. a) *. (s -. b) *. (s -. c))
```

```
let rec poly_area (ps : point list) : float =
  match ps with
  | p1 :: p2 :: p3 :: tail ->
    tri_area p1 p2 p3 +. poly_area (p1::p3::tail)
  | _ -> 0.
```

```
let area (s : shape) : float =
  match s with
  | Square s -> s *. s
  | Ellipse (r1, r2) -> pi *. r1 *. r2
  | RtTriangle (s1, s2) -> s1 *. s2 /. 2.
  | Polygon ps -> poly_area ps
```

TYPES DE DONNÉES INDUCTIFS

Types de données inductifs

- On peut utiliser des types de données pour définir des données inductives
- Un arbre binaire est:
 - Une feuille qui ne contient aucune donnée
 - un noeud qui contient une clé, une valeur, un sous-arbre de gauche et un sous-arbre de droite

Types de données inductifs

- On peut utiliser des types de données pour définir des données inductives
- Un arbre binaire est:
 - Une feuille qui ne contient aucune donnée
 - un noeud qui contient une clé, une valeur, un sous-arbre de gauche et un sous-arbre de droite

```
type key = int
type value = string

type tree =
  Leaf
  | Node of key * value * tree * tree
```

Types de données inductifs

```
type key = int
type value = string

type tree =
  Leaf
  | Node of key * value * tree * tree
```

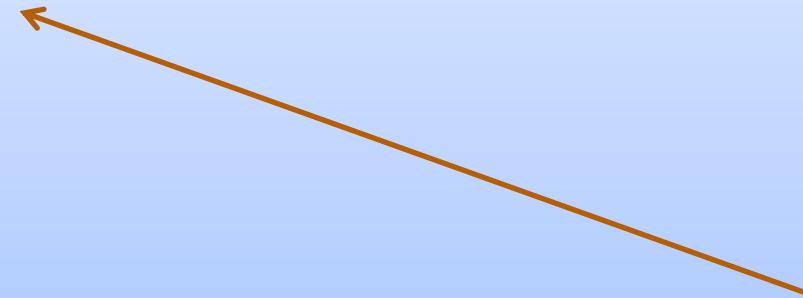
```
let rec insert (t:tree) (k:key) (v:value) : tree =
```

Types de données inductifs

```
type key = int
type value = string

type tree =
  Leaf
  | Node of key * value * tree * tree
```

```
let rec insert (t:tree) (k:key) (v:value) : tree =
  match t with
  | Leaf ->
  | Node (k', v', left, right) ->
```



Encore une fois, la définition de type détermine les cas qu'il faut inclure

Types de données inductifs

```
type key = int
type value = string

type tree =
  Leaf
  | Node of key * value * tree * tree
```

```
let rec insert (t:tree) (k:key) (v:value) : tree =
  match t with
  | Leaf -> Node (k, v, Leaf, Leaf)
  | Node (k', v', left, right) ->
```

Types de données inductifs

```
type key = int
type value = string

type tree =
  Leaf
  | Node of key * value * tree * tree
```

```
let rec insert (t:tree) (k:key) (v:value) : tree =
  match t with
  | Leaf -> Node (k, v, Leaf, Leaf)
  | Node (k', v', left, right) ->
    if k < k' then
      Node (k', v', insert left k v, right)
    else if k > k' then
      Node (k', v', left, insert right k v)
    else
      Node (k, v, left, right)
```

Types de données inductifs

```
type key = int
type value = string

type tree =
  Leaf
  | Node of key * value * tree * tree
```

```
let rec insert (t:tree) (k:key) (v:value) : tree =
  match t with
  | Leaf -> Node (k, v, Leaf, Leaf)
  | Node (k', v', left, right) ->
    if k < k' then
      Node (k', v', insert left k v, right)
    else if k > k' then
      Node (k', v', left, insert right k v)
    else
      Node (k, v, left, right)
```

Types de données inductifs

```
type key = int
type value = string

type tree =
  Leaf
  | Node of key * value * tree * tree
```

```
let rec insert (t:tree) (k:key) (v:value) : tree =
  match t with
  | Leaf -> Node (k, v, Leaf, Leaf)
  | Node (k', v', left, right) ->
    if k < k' then
      Node (k', v', insert left k v, right)
    else if k > k' then
      Node (k', v', left, insert right k v)
    else
      Node (k, v, left, right)
```

Types de données inductifs: autre exemple

- On peut utiliser le type "int" pour représenter des nombres naturels
 - mais ce type contient aussi des nombres négatifs
 - il faut utiliser un test dynamique et une valeur par défaut:

```
let double (n : int) : int =
  if n < 0 then
    0
  else
    double_nat n
```

Types de données inductifs: autre exemple

- On peut utiliser le type "int" pour représenter des nombres naturels
 - mais ce type contient aussi des nombres négatifs
 - il faut utiliser un test dynamique et une valeur par défaut:
 - ou lever une exception:

```
let double (n : int) : int =
  if n < 0 then
    raise (Failure "negative input!")
  else
    double_nat n
```

- il serait préférable de pouvoir définir **exactement** les nombres naturels et d'utiliser le système de types d'OCaml pour garantir qu'aucun client ne tente jamais d'appeler la fonction avec une valeur d'entrée négative

Types de données inductifs

- un nombre naturel n a la forme :
 - soit zero
 - soit $m + 1$
- On utilise un type de données pour représenter cette définition exactement :

Types de données inductifs

- un nombre naturel n a la forme :
 - soit zero
 - soit m + 1
- On utilise un type de données pour représenter cette définition exactement :

```
type nat = Zero | Succ of nat
```

Types de données inductifs

- un nombre naturel n a la forme :
 - soit zero
 - soit m + 1
- On utilise un type de données pour représenter cette définition exactement :

```
type nat = Zero | Succ of nat

let rec nat_to_int (n : nat) : int =
  match n with
    Zero -> 0
  | Succ n -> 1 + nat_to_int n
```

Types de données inductifs

- un nombre naturel n a la forme :
 - soit zero
 - soit m + 1
- On utilise un type de données pour représenter cette définition exactement :

```
type nat = Zero | Succ of nat

let rec nat_to_int (n : nat) : int =
  match n with
    Zero -> 0
  | Succ n -> 1 + nat_to_int n

let rec double_nat (n : nat) : nat =
  match n with
    | Zero -> Zero
    | Succ m -> Succ (Succ (double_nat m))
```

Résumé

- Types de données OCaml: un mécanisme puissant pour définir des structures de données complexes:
 - Ils sont précis
 - Ils contiennent exactement les éléments que vous voulez, pas plus
 - Ils sont généraux
 - récursif, non récursif (mutuellement récursif et polymorphe)
 - Le contrôle de type permet de détecter les erreurs
 - Il rapporte les cas manquants dans vos fonctions

DÉFINITIONS DE TYPE PARAMÉTRÉES

```
type ('key, 'val) tree =  
  Leaf  
  | Node of 'key * 'val * ('key, 'val) tree * ('key, 'val) tree
```

```
type 'a inttree = (int, 'a) tree
```

```
type istree = string inttree
```

Forme générale:

définition:

```
type 'x f = body
```

utilisation:

```
arg f
```

Une notation plus conventionnelle aurait été la suivante (mais cela n'est pas utilisé dans ML):

définition:

```
type f x = body
```

utilisation:

```
f arg
```

À retenir

- Pensez aux types paramétrés comme les fonctions:
 - une fonction qui prend un type en argument
 - et génère un type en sortie
- Base théorique:
 - Système F-Oméga
 - un lambda calcul typé avec des fonctions sur les types ainsi que des fonctions sur les valeurs