

Type definition exercise

- Given this new type

```
type PersonalInfo = Address of int * string  
                  | Phone of string | Age of int
```

define function `street: PersonalInfo → string` **that returns**
the street name for an address, and the empty string for any
other kind of value:

let `street pi = match pi with`
 `Address(i,s) → s`
 `| Phone s → ""`
 `| Age i → ""` } or `Address(_,s) → s`
 `| _ → ""`

Recursive type definitions

- In this type definition:

$$\text{type } t = C_1 [\text{of } te_1] \mid \dots \mid C_n [\text{of } te_n]$$

the type expressions te_i can contain t , making the type declaration recursive. This allows for the definition of infinite data types, such as lists and trees, e.g.

```
type mylist = Empty | Cons of int * mylist
let list1 = Cons (3, Cons (4, Empty))
```

- Ex: write the function $\text{sum} : \text{mylist} \rightarrow \text{int}$.

let rec sum lis = match lis with
 Empty $\rightarrow 0$
 | Cons (i, l) $\rightarrow i + \text{sum } l$

Exercises: Functions on binary trees

```
type bintree = Empty  
              | Node of int * bintree * bintree
```

- Define **isLeaf**: **bintree** $\rightarrow$ **bool**

let isLeaf t = match t with
 Node(-, Empty, Empty) $\rightarrow$ true
 | _ $\rightarrow$ false

- Define **sum**: **bintree** $\rightarrow$ **int**

let rec sum t $\rightarrow$ match t with
 Empty $\rightarrow$ 0
 | Node(i, t1, t2) $\rightarrow$ i + sum t1
 + sum t2