Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC
http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Variants - Syntax (slightly simplified)
- type name = C₁ [of ty₁] | . . . | Cₙ [of tyₙ]
- Introduce a type called name
- (fun x -> Cᵢ x) : tyᵢ -> name
- Cᵢ is called a constructor; if the optional type argument is omitted, it is called a constant
- Constructors are the basis of almost all pattern matching

Enumeration Types as Variants
An enumeration type is a collection of distinct values

In C and Ocaml they have an order structure; order by order of input

Functions over Enumerations

```
# type weekday = Monday | Tuesday | Wednesday 
    | Thursday | Friday | Saturday | Sunday;;
type weekday = 
    Monday 
  | Tuesday 
  | Wednesday 
  | Thursday 
  | Friday 
  | Saturday 
  | Sunday
```

```
# let rec days_later n day = 
    match n with 0 -> day 
    | _ -> if n > 0 ... (days_later (n - 1) day) 
         else days_later (n + 7) day;; 
val days_later : int -> weekday -> weekday = <fun>
```
Functions over Enumerations

```ml
# days_later 2 Tuesday;;
- : weekday = Thursday
# days_later (-1) Wednesday;;
- : weekday = Tuesday
# days_later (-4) Monday;;
- : weekday = Thursday
```

Problem:

```ml
# type weekday = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday;;

- Write function is_weekend : weekday -> bool
  let is_weekend day =
  match day with Saturday -> true
  | Sunday -> true
  | _ -> false
```

Example Enumeration Types

```ml
# type bin_op = IntPlusOp | IntMinusOp | EqOp | CommaOp | ConsOp

# type mon_op = HdOp | TlOp | FstOp | SndOp
```

Disjoint Union Types

- Disjoint union of types, with some possibly occurring more than once

```ml
# type id = DriversLicense of int | SocialSecurity of int | Name of string;;

val check_id : id -> bool = <fun>
```

Disjoint Union Types

- We can also add in some new singleton elements
Problem

Create a type to represent the currencies for US, UK, Europe and Japan

type currency =
   Dollar of int
 | Pound of int
 | Euro of int
 | Yen of int

Example Disjoint Union Type

# type const =
      BoolConst of bool  
    | IntConst of int 
    | FloatConst of float 
    | StringConst of string  
    | NilConst 
    | UnitConst

   How to represent 7 as a const?
   Answer:  IntConst 7

Polymorphism in Variants

The type 'a option is gives us something to represent non-existence or failure

# type 'a option = Some of 'a | None;;
type 'a option = Some of 'a | None

Used to encode partial functions
Often can replace the raising of an exception

Functions producing option

# let rec first p list =
    match list with [ ] -> None 
   | (x::xs) -> if p x then Some x else first p xs;;
val first : ('a -> bool) -> 'a list -> 'a option = <fun>

# first (fun x -> x > 3) [1;3;4;2;5];;
- : int option = Some 4
# first (fun x -> x > 5) [1;3;4;2;5];;
- : int option = None
### Functions over option

```ocaml
# let result_ok r =
  match r with
  | None -> false
  | Some _ -> true;;
val result_ok : 'a option -> bool = <fun>

# result_ok (first (fun x -> x > 3) [1;3;4;2;5]);;
- : bool = true

# result_ok (first (fun x -> x > 5) [1;3;4;2;5]);;
- : bool = false
```

### Problem

- Write a `hd` and `tl` on lists that doesn't raise an exception and works at all types of lists.

```ocaml
let hd list =
  match list with
  | [] -> None
  | (x::xs) -> Some x

let tl list =
  match list with
  | [] -> None
  | (x::xs) -> Some xs
```

### Mapping over Variants

```ocaml
# let optionMap f opt =
  match opt with
  | None -> None
  | Some x -> Some (f x);;
val optionMap : ('a -> 'b) -> 'a option -> 'b option = <fun>

# optionMap
  (fun x -> x - 2)
  (first (fun x -> x > 3) [1;3;4;2;5]);;
- : int option = Some 2
```

### Folding over Variants

```ocaml
# let optionFold someFun noneVal opt =
  match opt with
  | None -> noneVal
  | Some x -> someFun x;;
val optionFold : ('a -> 'b) -> 'b -> 'a option -> 'b option = <fun>

# optionMap
  (fun x -> x - 2)
  (first (fun x -> x > 3) [1;3;4;2;5]);;
- : int option = Some 2
```

### Recursive Types

- The type being defined may be a component of itself.
Recursive Data Types

```ocaml
# type exp =  
     VarExp of string 
   | ConstExp of const 
   | MonOpAppExp of mon_op * exp 
   | BinOpAppExp of bin_op * exp * exp 
   | IfExp of exp * exp * exp 
   | AppExp of exp * exp 
   | FunExp of string * exp
```

9/23/14 29

Recursive Data Types

```ocaml
# type bin_op = IntPlusOp | IntMinusOp  
          | EqOp | CommaOp | ConsOp | ... 
# type const = BoolConst of bool | IntConst of int | ...

# type exp = VarExp of string | ConstExp of const 
   | BinOpAppExp of bin_op * exp * exp 

- How to represent 6 as an exp?
  - Answer: ConstExp (IntConst 6)

9/23/14 30

Recursive Data Types

```ocaml
# type bin_op = IntPlusOp | IntMinusOp  
          | EqOp | CommaOp | ConsOp | ... 
# type const = BoolConst of bool | IntConst of int | ...
# type exp = VarExp of string | ConstExp of const 
   | BinOpAppExp of bin_op * exp * exp 

- How to represent (6, 3) as an exp?

9/23/14 31

Recursive Data Types

```ocaml
# type bin_op = IntPlusOp | IntMinusOp  
          | EqOp | CommaOp | ConsOp | ... 
# type const = BoolConst of bool | IntConst of int | ...
# type exp = VarExp of string | ConstExp of const 
   | BinOpAppExp of bin_op * exp * exp 

- How to represent ((6, 3)) as an exp?
  - BinOpAppExp (ConsOp, BinOpAppExp (CommaOp, ConstExp (IntConst 6), ConstExp (IntConst 3)), ConstExp NilCons)));
```

9/23/14 32
Your turn now

Try Problem 1 on MP5

Recursive Data Types

```ocaml
# type int_Bin_Tree = Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree);;

val int_Bin_Tree = Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree)
```

Recursive Data Type Values

```ocaml
# let bin_tree = Node(Node(Leaf 3, Leaf 6), Leaf (-7));;

val bin_tree : int_Bin_Tree = Node (Node (Leaf 3, Leaf 6), Leaf (-7))
```

Recursive Functions

```ocaml
# let rec first_leaf_value tree = 
  match tree with (Leaf n) -> n 
  | Node (left_tree, right_tree) -> first_leaf_value left_tree;;

val first_leaf_value : int_Bin_Tree -> int = <fun>

# let left = first_leaf_value bin_tree;;

val left : int = 3
```

Problem

```ocaml
# type int_Bin_Tree = Leaf of int 
  | Node of (int_Bin_Tree * int_Bin_Tree);;

- Write sum_tree : int_Bin_Tree -> int
- Adds all ints in tree

let rec sum_tree t =
```
**Problem**

```plaintext
type int_Bin_Tree = Leaf of int  
| Node of (int_Bin_Tree * int_Bin_Tree);;

Write sum_tree : int_Bin_Tree -> int

Adds all ints in tree
let rec sum_tree t =
    match t with
    | Leaf n -> n
    | Node(t1,t2) -> sum_tree t1 + sum_tree t2
```

**Mapping over Recursive Types**

```plaintext
# let rec ibtreeMap f tree =
    match tree with
    | Leaf n -> Leaf (f n)
    | Node (left_tree, right_tree) ->
        Node (ibtreeMap f left_tree, ibtreeMap f right_tree);

val ibtreeMap : (int -> int) -> int_Bin_Tree -> int_Bin_Tree = <fun>
```

```plaintext
# ibtreeMap ((+) 2) bin_tree;;
- : int_Bin_Tree = Node (Node (Leaf 5, Leaf 8), Leaf (-5))
```

**Folding over Recursive Types**

```plaintext
# let rec ibtreeFoldRight leafFun nodeFun tree =
    match tree with
    | Leaf n -> leafFun n
    | Node (left_tree, right_tree) ->
        nodeFun
            (ibtreeFoldRight leafFun nodeFun left_tree)
            (ibtreeFoldRight leafFun nodeFun right_tree);

val ibtreeFoldRight : (int -> 'a) -> ('a -> 'a -> 'a) ->
int_Bin_Tree -> 'a = <fun>
```

```plaintext
# let tree_sum =
    ibtreeFoldRight (fun x -> x) (+);;

val tree_sum : int_Bin_Tree -> int = <fun>
```

```plaintext
# tree_sum bin_tree;;
- : int = 2
```