Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC
https://courses.engr.illinois.edu/cs421/sp2023

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha
Variants - Syntax (slightly simplified)

- type \( \text{name} = C_1 [\text{of } \text{ty}_1] | \ldots | C_n [\text{of } \text{ty}_n] \)
- Introduce a type called \( \text{name} \)
- \((\text{fun} \ x \to C_i x) : \text{ty}_1 \to \text{name}\)
- \(C_i\) is called a \textit{constructor}; if the optional type argument is omitted, it is called a \textit{constant}
- Constructors are the basis of almost all pattern matching
Data type in Ocaml: lists

- Frequently used lists in recursive program
- Matched over two structural cases
  - `[ ]` - the empty list
  - `(x :: xs)` a non-empty list
- Covers all possible lists
- type `'a list = [ ] | (:::) of 'a * 'a list`
  - Not quite legitimate declaration because of special syntax
An enumeration type is a collection of distinct values.

In C and Ocaml they have an order structure; order by order of input.
# type weekday = Monday | Tuesday | Wednesday
| Thursday | Friday | Saturday | Sunday;;

type weekday =
    Monday
| Tuesday
| Wednesday
| Thursday
| Friday
| Saturday
| Sunday
Functions over Enumerations

# let day_after day = match day with
| Monday -> Tuesday
| Tuesday -> Wednesday
| Wednesday -> Thursday
| Thursday -> Friday
| Friday -> Saturday
| Saturday -> Sunday
| Sunday -> Monday;;

val day_after : weekday -> weekday = <fun>
# let rec days_later n day =
match n with 0 -> day
| _ -> if n > 0
    then day_after (days_later (n - 1) day)
else days_later (n + 7) day;;

val days_later : int -> weekday -> weekday
  = <fun>
Functions over Enumerations

# days later 2 Tuesday;
- : weekday = Thursday

# days later (-1) Wednesday;
- : weekday = Tuesday

# days later (-4) Monday;
- : weekday = Thursday
# Problem:

Write function `is_weekend : weekday -> bool`.

```plaintext
let is_weekend day =
```

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

```ocaml
# 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

```ocaml
type bin_op = IntPlusOp | IntMinusOp | EqOp | CommaOp | ConsOp
```

```ocaml
type mon_op = HdOp | TlOp | FstOp | SndOp
```
Disjoint Union Types

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

- We can also add in some new singleton elements
Disjoint Union Types

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

# type id = DriversLicense of int | SocialSecurity of int | Name of string

# let check_id id = match id with
  DriversLicense num ->
      not (List.mem num [13570; 99999])
  | SocialSecurity num -> num < 900000000
  | Name str -> not (str = "John Doe");

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

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

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

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

```ocaml
# 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

```ocaml
# 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.
Problem

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

- `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 = <fun>

# let optionMap f opt =
  optionFold (fun x -> Some (f x)) None opt;;
val optionMap : ('a -> 'b) -> 'a option -> 'b option = <fun>
```
Recursive Types

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

# type int_Bin_Tree =

Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree);


type int_Bin_Tree = Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree)
Recursive Data Type Values

# 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 Data Type Values

\[
\text{bin\_tree} = \text{Node} \quad \text{Leaf} \ (-7) \\
\text{Node} \quad \text{Leaf} 3 \quad \text{Leaf} 6
\]
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
```
Recursive Data Types

# 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
Recursive Data Types

# 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?
Recursive Data Types

```haskell
# 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)`
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?
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 (CommaOp, ConstExp (IntConst 6), ConstExp (IntConst 3))
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?
Problem

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 =
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
Recursion over Recursive Data Types

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

- How to count the number of variables in an exp?
Recursion over Recursive Data Types

```haskell
# type exp = VarExp of string | ConstExp of const
  | BinOpAppExp of bin_op * exp * exp
  | FunExp of string * exp | AppExp of exp * exp

How to count the number of variables in an exp?

# let rec varCnt exp =
  match exp with VarExp x ->
    | ConstExp c ->
    | BinOpAppExp (b, e1, e2) ->
    | FunExp (x,e) ->
    | AppExp (e1, e2) ->
```
Recursion over Recursive Data Types

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

- How to count the number of variables in an exp?

# let rec varCnt exp =
  match exp with
    VarExp x -> 1
    ConstExp c -> 0
    BinOpAppExp (b, e1, e2) -> varCnt e1 + varCnt e2
    FunExp (x,e) -> 1 + varCnt e
    AppExp (e1, e2) -> varCnt e1 + varCnt e2
Mapping over Recursive Types

```ocaml
# 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>
```
Mapping over Recursive Types

# ibtreeMap ((+) 2) bin_tree;;

- : int_Bin_Tree = Node (Node (Leaf 5, Leaf 8), Leaf (-5))
Folding over Recursive Types

```ocaml
# 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>
```
Folding over Recursive Types

# let tree_sum =
   ibtreeFoldRight (fun x -> x) (+);;
val tree_sum : int_Bin_Tree -> int = <fun>
# tree_sum bin_tree;;
- : int = 2