Lecture 35:

- Quiz 8
- OCaml: Algebraic Types

Announcements:

- HW-7 due
- HW-8 out (soon)

OCaml User-Defined Types

Algebraic Types: In type theory, "algebraic" types consist of ...

- product types ... i.e., a cartesian product of types $t_1 \times t_2 \times \cdots \times t_n$
- sum types ... i.e., a disjoint unions of types $t_1 \sqcup t_2 \sqcup \cdots \sqcup t_n$
- combinations of these

OCaml product types (aka record types)

```
type book = Book of int * string * string list
```

- book is the **type name** ... not capitalized
- Book is a (value) **constructor** ... capitalized
- everything after Book is a **field**
- note in OCaml multiple fields represented as tuples (3-tuple above)

OCaml User-Defined Types

Using our new type ...

```
# let b1 = Book (1, "Compilers", ["Aho"; "Ullman"]) ;;
val b1 : book = Book (1, "Compilers", ["Aho"; "Ullman"])
# let b2 = Book (2, "Elements of ML", ["Ullman"]) ;;
val b2 : book = Book (2, "Elements of ML", ["Ullman"])
```

OCaml sum types (aka union types)

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

A sum type can have multiple value constructors ...

- note: value constructors can be defined with 0 or more fields ...
- here essentially a named value of a type
- like `true` is a named `bool` value

OCaml User-Defined Types

Using our new type ...

```
# let c1 = Red ;;
val c1 : color = Red
# let c2 = Green ;;
val c2 : color = Green
# c1 == c2 ;;
- : bool = false
# c2 == Green ;;
- : bool = true
# Red < Green && Green < Blue ;;
- : bool = true;
```

Each constructor can also have different fields ...

```
type periodical = Magazine of string * int
                | Newpaper of string * string * int

# Magazine ("Quanta", 112233) ;;
- : periodical = Magazine ("Wired", 112233)
# Newspaper ("Spokesman", "April", 2025) ;;
- : periodical = Newspaper ("Spokesman", "April", 2025)
```

OCaml User-Defined Types

Can define functions over algebraic types using pattern matching

```
let color_string c =
  match c with
  | Red -> "red"
  | Green -> "green"
  | Blue -> "blue"

# color_string ;;
val color_string : color -> string = <fun>

let title p =
  match p with
  | Magazine (t, _) -> t
  | Newspaper (t, _, _) -> t

# title ;;
val title : periodical -> string = <fun>
```


OCaml User-Defined Types

Parametric types: algebraic types with type parameters

```
type 'a box = Box of 'a
            | EmptyBox
```

A box is like an optional – just a container for a value of any type

```
# let b1 = Box true ;;
val b1 : bool box = Box true

# let b2 = Box 42 ;;
val b2 : int box = Box 42

# let b3 = EmptyBox ;;
val b3 : 'a box = EmptyBox

# let b4 = Box [] ;;
val b4 : 'a list box = Box []
```

OCaml User-Defined Types

Example box function:

```
let div_boxes b1 b2 =
  match (b1, b2) with
  | (Box x, Box y) when y != 0 -> Box (x/y)
  | (_, _) -> EmptyBox

# div_boxes ;;
val div_boxes : int box -> int box -> int box = <fun>

# div_boxes (Box 4) (Box 2) ;;
- : int box = Box 2

# div_boxes (Box 3) (Box 0) ;;
- : int box = Empty

# div_boxes (Box 2) EmptyBox ;;
- : int box = Empty
```


OCaml User-Defined Types

Linked lists via a recursive parameterized type

```
type 'a linked_list = Node of 'a * 'a linked_list
                    | Null
```

- A `Node` (value) consists of an (`'a` value, `linked_list`) pair
- `Null` is a list "terminator" value

Examples:

```
# let l1 = Node ("a", Node ("b", Node ("c",Null))) ;;
val l1 : string linked_list = Node ("a", Node ("b", Node ("c", Null)))

# let l2 = Node (1, Node (2, Node (3, Node (4, Null)))) ;;
val l2 : int link_list = Node (1, Node (2, Node (3, Node (4, Null))))
```

OCaml User-Defined Types

Check if a linked list is empty:

```
let empty xs =
   match xs with
   | Null -> true
   | _ -> false

# empty ;;
val empty : 'a linked_list -> int = <fun>
```

Compute the length of a linked list

```
let rec length xs =
   match xs with
   | Null -> 0
   | Node (_, tail) -> 1 + length tail

# length ;;
val length : 'a linked_list -> int = <fun>
```