Products, Unions, Algebraic Data Types

Concepts of Programming Languages
Lecture 7

CAS CS 320

Practice Problem

$$\{ \texttt{x} : \texttt{int}, \texttt{y} : \texttt{int} \} \vdash \texttt{(fun z -> y) x} : \texttt{int}$$

Write a derivation of the above typing judgment the inference rules of 320Caml

Outline

- Discuss the use of **tuples** and **records** for creating collections of data
- Introduce **algebraic data types (ADTs)** for creating data with given "shapes"
- Cover **parametric** and **recursive** ADTs for more general data structures

Products

Tuples

```
let point : float * float = (2.0, 3.0)
let student : string * int = ("Franco", 244342)
```

Tuples

```
let point : float * float = (2.0, 3.0)
let student : string * int = ("Franco", 244342)
```

ordered unlabeled fixed-length heterogeneous
collections of data

Tuples

```
let point : float * float = (2.0, 3.0)
let student : string * int = ("Franco", 244342)
```

ordered unlabeled fixed-length heterogeneous
collections of data

Primarily useful for "returning" multiple arguments

Pattern Matching on Tuples

```
let hypotenuse (p : float * float) : float =
  match p with
  | (x, y) -> sqrt (x *. x +. y *. y)
```

There are no accessors for tuples

Instead we use **pattern matching**

Pattern Matching in General

```
match e with
| p -> o
| ...
```

Pattern Matching in General

```
match e with
| p -> o
| ...
```

A **pattern** is a typed template for how a piece of data should look

Pattern Matching in General

```
match e with
| p -> o
| ...
```

A **pattern** is a typed template for how a piece of data should look

A **match-expression** is a way of *destructing* data in OCaml

Pattern Matching in General

```
match e with
| p -> o
| ...
```

A **pattern** is a typed template for how a piece of data should look

A **match-expression** is a way of *destructing* data in OCaml

We *match* on an expression e, and check if the value of e *matches* with the pattern p

Note: Patterns are not Expressions

```
<expr> ::= match <expr> with
         | <pattern> -> <expr>
         | <pattern> -> <expr>
         | ....
```

Patterns are similar to expressions, but not identical

They can be wildcards, they can be variables, there's a lot of possibilities (link)

Advanced Pattern Matching

```
let hypotenuse ((x, y) : float * float) : float =
  sqrt (x *. x +. y *. y)

let hypotenuse (p : float * float) : float =
  let (x, y) = p in
  sqrt (x *. x +. y *. y)
```

Pattern matching can also be done implicitly in let-expressions and function arguments

demo

And we can do all this formally!

Tuples (Syntax Rule)

<expr> ::= <expr> , ... , <expr>

If $e_1, \ldots, e_n$ are well-formed expressions, then

$$e_1 \text{ , ... , } e_n$$

is a well-formed expression

Tuple (Typing Rule)

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash e_1 \ , \ \ldots \ , \ e_n : \tau_1 \ * \ \ldots \ * \ \tau_n} \text{ (tuple)}$$

If $e_1, \ldots, e_n$ are of type $\tau_1, \ldots, \tau_n$, respectively, in the context Γ then

$$e_1 \ , \ \ldots \ , \ e_n$$

is of type $\tau_1 * \ldots * \tau_n$ in the context Γ

Tuple (Semantic Rule)

$$\frac{e_1 \Downarrow v_1 \qquad \ldots \qquad e_n \Downarrow v_n}{e_1 \ , \ \ldots \ , \ e_n \Downarrow (v_1, \ldots, v_n)} \text{ (tupleE)}$$

If $e_1, \ldots, e_n$ evaluate to $v_1, \ldots, v_n$, respectively, then

$$(\ e_1 \ , \ \ldots \ , \ e_n \ )$$

evaluates to $(v_1, \ldots, v_n)$

Example (Typing Derivation)

$$\{x : \texttt{int}\} \vdash (\ x + x\ ,\ \texttt{true}\ ) : \texttt{int} * \texttt{bool}$$

Example (Semantic Derivation)

$$\texttt{(2 + 3 , true)} \Downarrow (5, \mathsf{T})$$

Pair Match (Typing Rule)

$$\frac{\Gamma \vdash p : \tau_1 \ * \ \tau_2 \qquad \Gamma, x : \tau_1, y : \tau_2 \vdash e : \tau}{\Gamma \vdash \texttt{match}\ p\ \texttt{with}\ \texttt{|}\ x\ \texttt{,}\ y\ \texttt{->}\ e : \tau}\ \text{(pairMatch)}$$

If p is of type $\tau_1 * \tau_2$ in the context Γ and e is of type τ in the context Γ along with x begin type τ_1 and y begin type τ_2 then

$$\texttt{match}\ p\ \texttt{with}\ \texttt{|}\ x\ \texttt{,}\ y\ \texttt{->}\ e$$

is of type τ in the context Γ

Pair Match (Semantic Rule)

$$\frac{\texttt{(}\ e_1\ \texttt{,}\ e_2\ \texttt{)} \Downarrow (v_1, v_2) \qquad [v_2/y][v_1/x]e \Downarrow v}{\texttt{match}\ p\ \texttt{with}\ \texttt{|}\ x\ \texttt{,}\ y\ \texttt{->}\ e \Downarrow v}\ \text{(pairMatch)}$$

If (e_1 , e_2) evaluates to (v_1, v_2) and e evaluates to v after substituting v_1 for x and v_2 for y, then

match p with | x , y -> e

evaluates to v

Practice Problem

$$\emptyset \vdash \texttt{fun x -> match x with | (a, b) -> a + b} : \tau$$

Determine the type τ so that the above judgment is derivable from the rules below. Also give a derivation

$$\frac{\Gamma \vdash p : \tau_1 \texttt{ * } \tau_2 \qquad \Gamma, x : \tau_1, y : \tau_2 \vdash e : \tau}{\Gamma \vdash \texttt{match } p \texttt{ with | } x \texttt{ , } y \texttt{ -> } e : \tau} \text{ (tuple)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ + } e_2 : \texttt{int}} \text{ (addInt)}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun } x \texttt{ -> } e : \tau_1 \texttt{ -> } \tau_2} \text{ (fun)}$$

$$\frac{(v : \tau) \in \Gamma}{\Gamma \vdash v : \tau} \text{ (intLit)}$$

$\emptyset \vdash$ `fun x -> match x with | (a, b) -> a + b :` τ

Solution

Records

```
type point = { x_cord : float ; y_cord : float }
let origin : point = { x_cord = 0. ; y_cord = 0. }

type user = {
  name : string ;
  email : string ;
  num_posts : int ;
}
```

unordered **labeled** **fixed-length** **heterogeneous** collections of data

They are useful for organizing large collections of data (akin to database records)

Record Syntax (Informal)

```
type record_ty =
  {
    field1 : ty1;
    field2 : ty2;
    ...
    fieldn : tyn;
  }
```

```
let record_expr : record_ty =
  {
    field1 = expr1;
    field2 = expr2;
    ...
    fieldn : exprn;
  }
```

For a record, we have to specify the type of each field

When we construct a record, every field must have a value

Accessors

```
type point = { x_cord : float ; y_cord : float }

let dist (p : point) (q : point) =
  let xd = p.x_cord -. q.x_cord in
  let yd = p.y_cord -. q.y_cord in
  sqrt (xd *. xd +. yd *. yd)
```

Records support **dot-notation**

(we can also access records by pattern matching)

Record Updates

```
let new_post u : user =
  { u with num_posts = u.num_posts + 1 }
```

Record Updates

```
let new_post u : user =
  { u with num_posts = u.num_posts + 1 }
```

We can use **with-syntax** to update a smaller number of fields in a large record

Record Updates

```
let new_post u : user =
  { u with num_posts = u.num_posts + 1 }
```

We can use **with-syntax** to update a smaller number of fields in a large record

"***u*** *with* ***num_posts*** *incremented, keep everything else the same*"

Record Updates

```
let new_post u : user =
  { u with num_posts = u.num_posts + 1 }
```

We can use **with-syntax** to update a smaller number of fields in a large record

"***u*** *with* ***num_posts*** *incremented, keep everything else the same*"

Data in functional languages are immutable. This returns a new record with the update

Practice Problem

Write the syntax, typing and semantic rules for records

demo

Unions

Simple Variants

```
type os = BSD | Linux | MacOS | Windows
```

A **simple variant** is a *user-defined* type for values of a fixed collection of possibilities

Type names are **lower_case** and Constructors names are **Upper_case**

Simple Variants

```
                constructor
type os = BSD | Linux | MacOS | Windows
```

A **simple variant** is a *user-defined* type for values of a fixed collection of possibilities

Type names are **lower_case** and Constructors names are **Upper_case**

Pattern Matching

```
let supported (sys : os) : bool =
  match sys with
  | BSD -> false
  | _ -> true
```

We work with variants by **pattern matching:**

» give a **pattern** that a value can **match** with

» give an output value for each pattern

Pattern Matching

```
let supported (sys : os) : bool =
  match sys with
| BSD -> false
| _ -> true
```

constant pattern

wildcard pattern

We work with variants by **pattern matching:**

» give a **pattern** that a value can **match** with

» give an output value for each pattern

Data-Carrying Variants

```
type linux_distro = Arch | Fedora | NixOS | Ubuntu

type os
  = BSD of int * int
  | Linux of linux_distro * int
  | MacOS of int
  | Windows of int

let supported (sys : os) : bool =
  match sys with
  | BSD (major , minor) -> major > 2 && minor > 3
  | _ -> true
```

Variants can carry data, which allows us to represent more complex structures

Data-Carrying Variants

```
type linux_distro = Arch | Fedora | NixOS | Ubuntu

type os
  = BSD of int * int
  | Linux of linux_distro * int
  | MacOS of int
  | Windows of int

let supported (sys : os) : bool =
  match sys with
  | BSD (major , minor) -> major > 2 && minor > 3
  | _ -> true
```

Note the syntax

Variants can carry data, which allows us to represent more complex structures

Aside: Constructor Arguments are not Tuples

```
type t = A of int * int
let args : int * int = (2, 3)
(* let a : t = A args *)
```

This code (uncommented) won't type-check

Arguments need to passed in directly

Aside: Constructors are not function

```
type t = A of int
let apply (f : int -> t) (x : int) : t = f x
(* let x : int = apply A t *)
```

This code (uncommented) won't type check

We **cannot** partially apply constructors

Aside: Constructors are not function

```
type t = A of int
let apply (f : int -> t) (x : int) : t = f x
(* let x : int = apply A t *)
```

function as an argument

This code (uncommented) won't type check

We **cannot** partially apply constructors

Pro Tip: Named Data-Carrying Variants

```
type os
  = MacOS of { major : int; minor : int; patch : int }
  | ...

let support (sys : os) : bool =
  match sys with
  | MacOS info ->
    info.major = 10 && info.minor >= 14 && info.patch >= 1
    (* MacOS Sonoma 10.14.(1-3) *)
  |...
```

Since we can carry *any* kind of data in a constructor, we can carry **records** to **name the parts** of our carried data

Practice Problem

```
let area (s : shape) =
  match s with
  | Rect r -> r.base *. r.height
  | Triangle { sides = (a, b) ; angle } -> Float.sin angle *. a *. b
  | Circle r -> r *. r *. Float.pi
```

*Define the variant **shape** which makes this function type-check*

What about variable length data?

Algebraic Data Types

Recursive Data

```
type t
  = A
  | B of t
  | C of t * t

let x : t = B (C (A, B A))
```

A variant type **t** can carry data of type **t**

Why would we want to do this?

Example: Lists

```
type intlist
  = Nil
  | Cons of int * intlist

let example = Cons (1, Cons (2, Cons (3, Nil)))
```

The type **intlist** is available as the type of data which a constructor of **intlist** holds

We can use recursive ADTs to create variable-length data types

Example: Lists

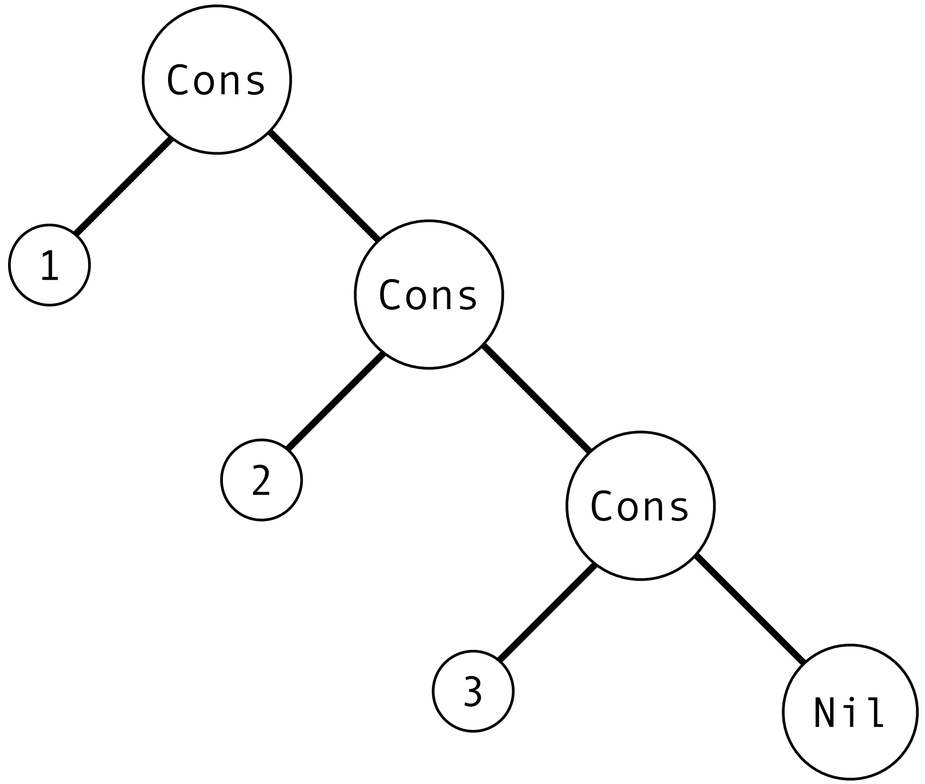
```
type intlist
  = Nil
  | Cons of int * intlist

let example = Cons (1, Cons (2, Cons (3, Nil)))
```

The type **intlist** is available as the type of data which a constructor of **intlist** holds

We can use recursive ADTs to create variable-length data types

The Picture

```
Cons (1,
  Cons (2,
    Cons (3,
      Nil)))
```



We think of values of recursive variants as trees with constructors as nodes and carried data as leaves

A More Interesting Example: Expressions

$$3 + ((2 * 4) - 14)$$

A More Interesting Example: Expressions

$$3 + ((2 * 4) - 14)$$

Suppose we're building a calculator...

A More Interesting Example: Expressions

$$3 + ((2 * 4) - 14)$$

Suppose we're building a calculator...

Before we compute the value of an input, it's useful to find an abstract representation of the input.

A More Interesting Example: Expressions

$$3 + ((2 * 4) - 14)$$

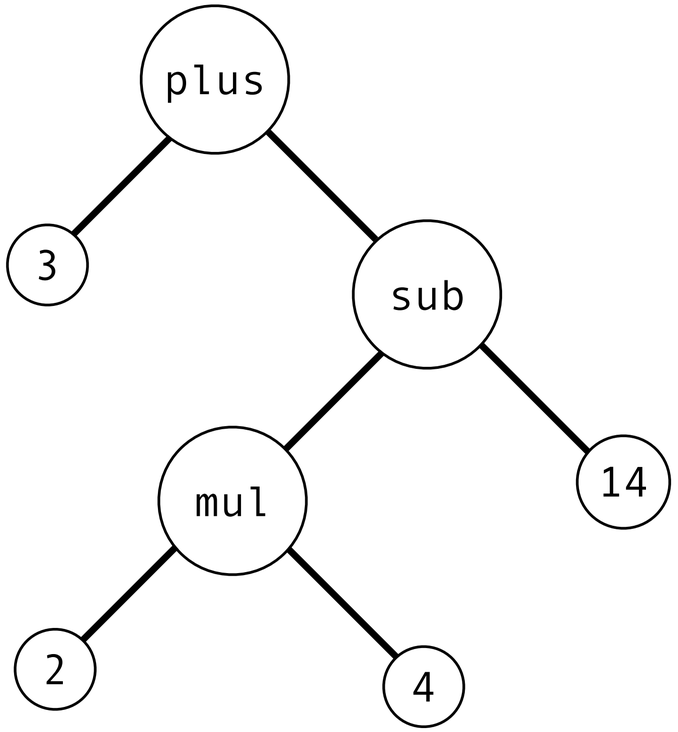
Suppose we're building a calculator...

Before we compute the value of an input, it's useful to find an abstract representation of the input.

This will help us separate the tasks of evaluation and parsing

A More Interesting Example: Expressions

$$3 + ((2 * 4) - 14) \Rightarrow$$



We can represent an expression abstractly as a tree with operations as nodes and number values as leaves

A More Interesting Example: Expressions

```
type expr
  = Val of int
  | Add of expr * expr
  | Sub of expr * expr
  | Mul of expr * expr

let _ = Add (Val 3, Sub (Mul (Val 2, Val 4), Val 14))
```

Which means we can represent it as a recursive variant!

We'll come back to this next time...

Parameterized Variants

```
type 'a mylist
  = Nil
  | Cons of 'a * 'a mylist

let e1 : int mylist = Cons (1, Cons (2, Cons (3, Nil)))
let e2 : string mylist = Cons ("1", Cons ("2", Cons ("3", Nil)))
```

Parameterized Variants

```
type 'a mylist
  = Nil
  | Cons of 'a * 'a mylist

let e1 : int mylist = Cons (1, Cons (2, Cons (3, Nil)))
let e2 : string mylist = Cons ("1", Cons ("2", Cons ("3", Nil)))
```

The last piece of the puzzle: variants can be type agnostic

Parameterized Variants

```
type 'a mylist
  = Nil
  | Cons of 'a * 'a mylist

let e1 : int mylist = Cons (1, Cons (2, Cons (3, Nil)))
let e2 : string mylist = Cons ("1", Cons ("2", Cons ("3", Nil)))
```

The last piece of the puzzle: variants can be type agnostic

This gives us a variant which is parametrically polymorphic

Parameterized Variants

```
       type variable
type 'a mylist
  = Nil
  | Cons of 'a * 'a mylist

let e1 : int mylist = Cons (1, Cons (2, Cons (3, Nil)))
let e2 : string mylist = Cons ("1", Cons ("2", Cons ("3", Nil)))
```

The last piece of the puzzle: variants can be type agnostic

This gives us a variant which is parametrically polymorphic

Parameterized Variants

type variable

```
type 'a mylist
  = Nil
  | Cons of 'a * 'a mylist
```

type constructor

```
let e1 : int mylist = Cons (1, Cons (2, Cons (3, Nil)))
let e2 : string mylist = Cons ("1", Cons ("2", Cons ("3", Nil)))
```

The last piece of the puzzle: variants can be type agnostic

This gives us a variant which is parametrically polymorphic

Recall: Parametric Polymorphism

```
let rev_tail (l : 'a list) : 'a list =
  let rec go acc l =
    match l with
    | [] -> acc
    | x :: xs -> go (x :: acc) xs
  in go [] l
```

Recall: Parametric Polymorphism

```
let rev_tail (l : 'a list) : 'a list =
  let rec go acc l =
    match l with
    | [] -> acc
    | x :: xs -> go (x :: acc) xs
  in go [] l
```

This allows us to write functions which can be more generally applied (reversing a list **does not depend on** what's in the list)

demo

Useful ADTs

Recall: Lists

```
type 'a list
  = []
  | (::) of 'a * 'a list
```

ordered **unlabeled** **variable-length** **homogeneous**
collections of data

Many important operations on data can be represented as operations on lists (e.g., updating all users in a database)

Options

```
type 'a option = None | Some of 'a

let head (l : 'a list) : 'a option =
  match l with
  | [] -> None
  | x :: xs -> Some x
```

Options

```
type 'a option = None | Some of 'a

let head (l : 'a list) : 'a option =
  match l with
  | [] -> None
  | x :: xs -> Some x
```

Options are like boxes which *may* hold a value or may be empty

Options

```
type 'a option = None | Some of 'a

let head (l : 'a list) : 'a option =
  match l with
  | [] -> None
  | x :: xs -> Some x
```

Options are like boxes which *may* hold a value or may be empty

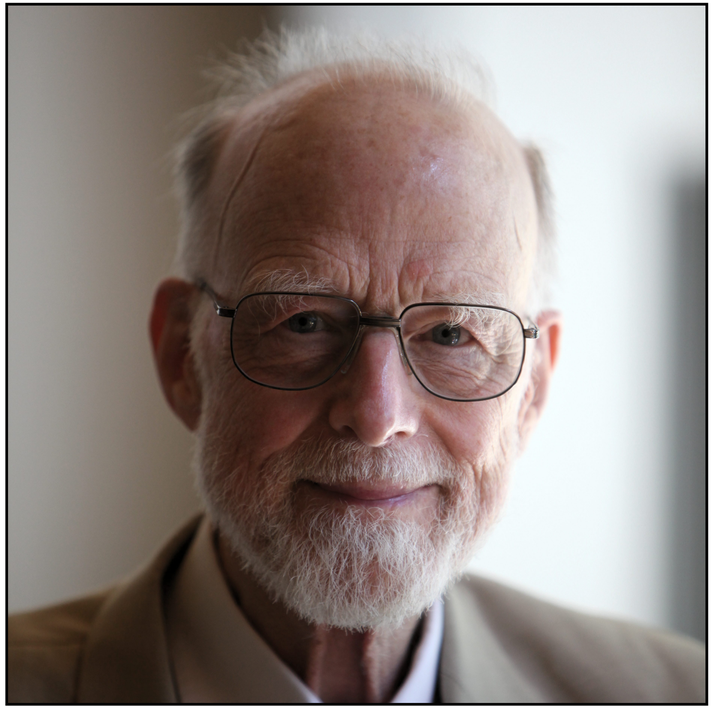
This can be useful for defining functions which **may not be total**

Aside: The Billion-Dollar Mistake

Tony Hoare calls his invention of the **null pointer** a "billion-dollar mistake"

OCaml doesn't have null pointers

I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.



Type-Driven Design

```
let head (l : 'a list) : 'a option =
  match l with
  | [] -> None
  | x :: _ -> Some(x)
```

Types should mirror the logic of our programs.
Then we take advantage of the type checker to verify our code (e.g., no reference to null)

Results

```
type ('a, 'e) result =
  | Ok of 'a
  | Error of 'e

let head (l : 'a list) : ('a, string) result =
  match l with
  | [] -> Error "[] has no first element"
  | x :: xs -> Ok x
```

A **result** is an option with additional data in the "None" case

Results

```
type ('a, 'e) result =
  | Ok of 'a
  | Error of 'e

let head (l : 'a list) : ('a, string) result =
  match l with
  | [] -> Error "[] has no first element"
  | x :: xs -> Ok x
```

Error message

A **result** is an option with additional data in the "None" case

Results

Note the syntax

```
type ('a, 'e) result =
  | Ok of 'a
  | Error of 'e

let head (l : 'a list) : ('a, string) result =
  match l with
  | [] -> Error "[] has no first element"
  | x :: xs -> Ok x
```

Error message

A **result** is an option with additional data in the "None" case

Aside: Built-in Variants

```
utop # #show List;;
module List :
  sig
    type 'a t = 'a list = [] | (::) of 'a * 'a list
    val length : 'a t -> int
    val compare_lengths : 'a t -> 'b t -> int
    val compare_length_with : 'a t -> int -> int
    val is_empty : 'a t -> bool
    val cons : 'a -> 'a t -> 'a t
    val hd : 'a t -> 'a
    ...
```

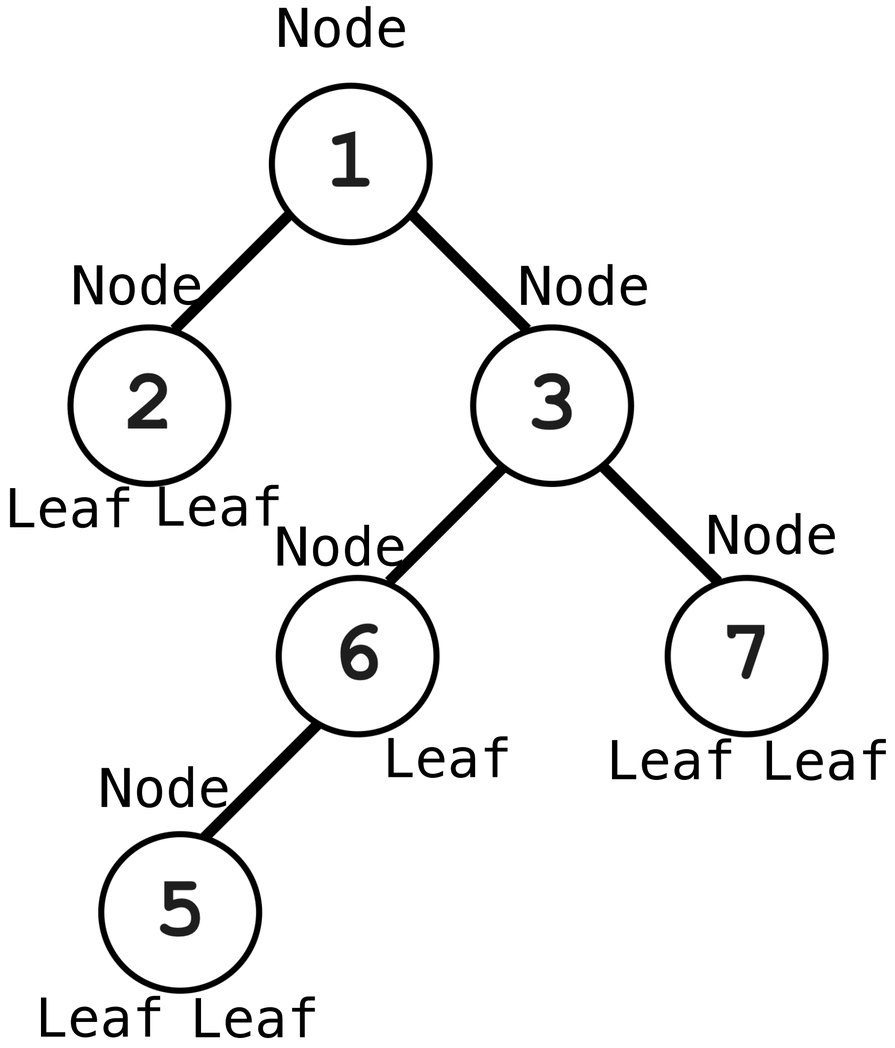
lists and optionals and results are built into OCaml

You can also use the **#show** directive in Utop to see the type signatures of functions available for lists, options and results.

Trees

```
type 'a tree =
  | Leaf
  | Node of 'a * 'a tree * 'a tree
```

A tree is a leaf with a value or a node with a left or right subtree



demo

Summary

Tuples, records, and **ADTs** help us organize data and create abstract interfaces

Recursive and **parametrized** ADTs give is richer structure