ADTs

Concepts of Programming Languages
Lecture 8

CAS CS 320

Announcements

» Quiz is next week on Tuesday in–class

» You should have gotten an email if you have accommodations or conflicts. If you have not, please talk to us immediately

» Hope you all saw the mock quiz. The actual quiz should be easier.

Practice Problem

int*int   int   int*int → int

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

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

$$\frac{\Gamma \vdash p : \tau_1 * \ldots * \tau_n \qquad \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau}{\Gamma \vdash \textsf{match}\ p\ \textsf{with}\ |\ x_1, \ldots, x_n \to e : \tau}\ (\textsf{matchTuple})$$

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

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \textsf{fun}\ x \to e : \tau_1 \to \tau_2}\ (\textsf{fun})$$

$$\frac{(v : \tau) \in \Gamma}{\Gamma \vdash v : \tau}\ (\textsf{var})$$

Solution

$$\dfrac{\dfrac{\dfrac{}{\{x : \text{int} * \text{int}\} \vdash x : \text{int} * \text{int}} \qquad \dfrac{\dfrac{}{\{x : \text{int} * \text{int}, a : \text{int}, b : \text{int}\} \vdash a : \text{int}} \quad \dfrac{}{\{x : \text{int} * \text{int}, a : \text{int}, b : \text{int}\} \vdash b : \text{int}}}{\{x : \text{int} * \text{int}, a : \text{int}, b : \text{int}\} \vdash a + b : \text{int}}}{\{x : \text{int} * \text{int}\} \vdash \text{match } x \text{ with } \mid (a, b) \to a + b : \text{int}}}{\emptyset \vdash \text{fun } x \to \text{match } x \text{ with } \mid (a, b) \to a + b : \text{int} * \text{int} \to \text{int}}$$

Did you do this?

- Define formal syntax rules for records
- Define formal typing rules for records
- Define formal semantics rules for records

Outline

- Introduce **algebraic data types (ADTs)** for creating data with given "shapes"
- Cover **parametric** and **recursive** ADTs for more general data structures
- Lots of examples today!

Unions

Second Greatest Feature of OCaml

Simple Variants

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

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

First letter of type names is **lower_case** and Constructor names is **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

First letter of type names is **lower_case** and Constructor names is **Upper_case**

Pattern Matching

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

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

» giving a <u>pattern</u> that a value can <u>match</u> with

» writing what to do for each pattern

Pattern Matching

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

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

» giving a pattern that a value can match with

» writing what to do 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

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?

Recursive ADTs

Greatest Feature of OCaml

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

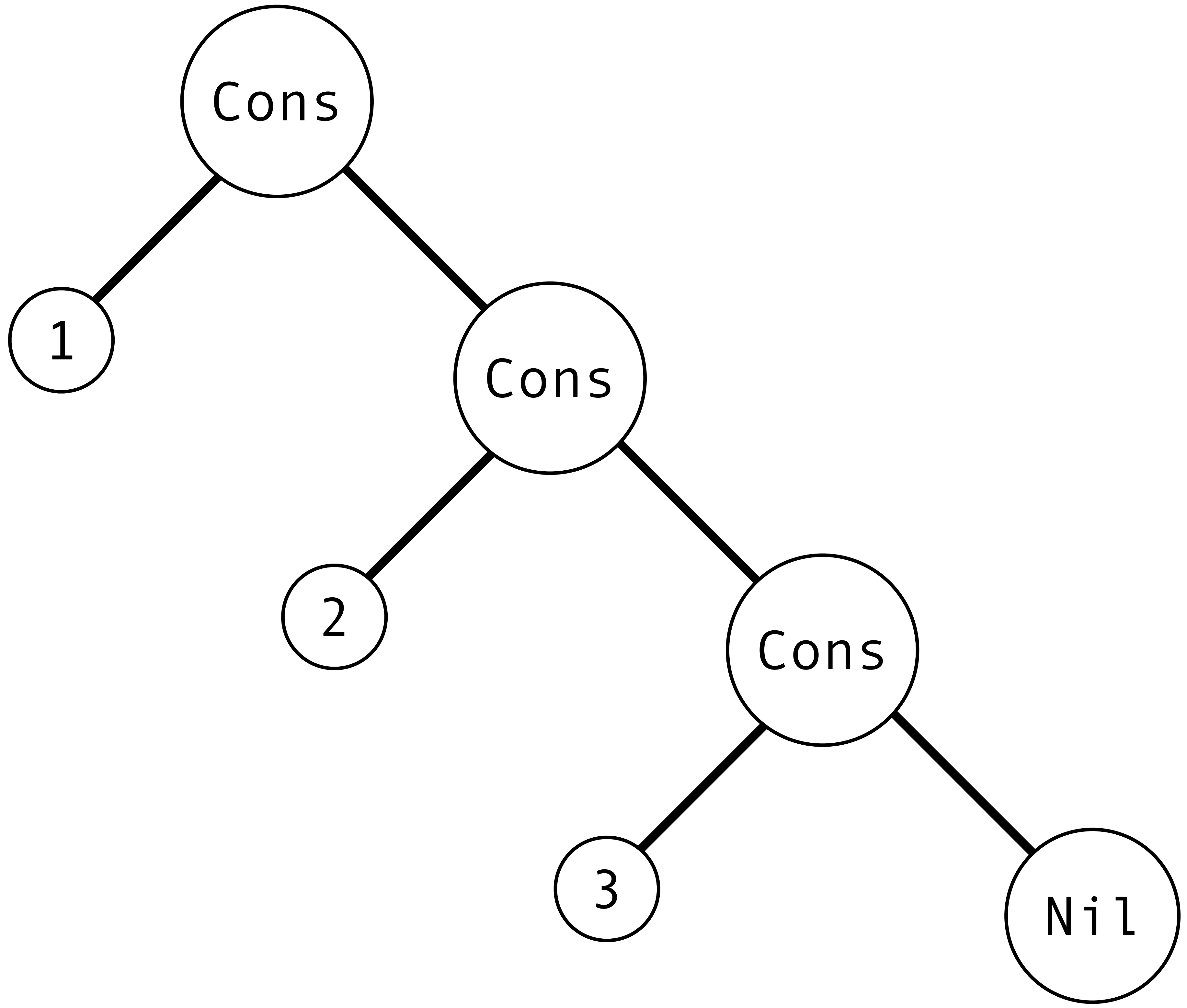
```
type intlist
  = Nil
  | Cons of int * intlist

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

demo

(basic functions for intlist)

Parametrized ADTs

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

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
```

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)

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)

Note. Because of **type-inference**, we rarely have to think about this

Useful ADTs

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.

Results

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

let head (l : 'a list) : ('a, string) myresult =
  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) myresult =
  | Ok of 'a
  | Error of 'e

let head (l : 'a list) : ('a, string) myresult =
  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) myresult =
  | Ok of 'a
  | Error of 'e

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

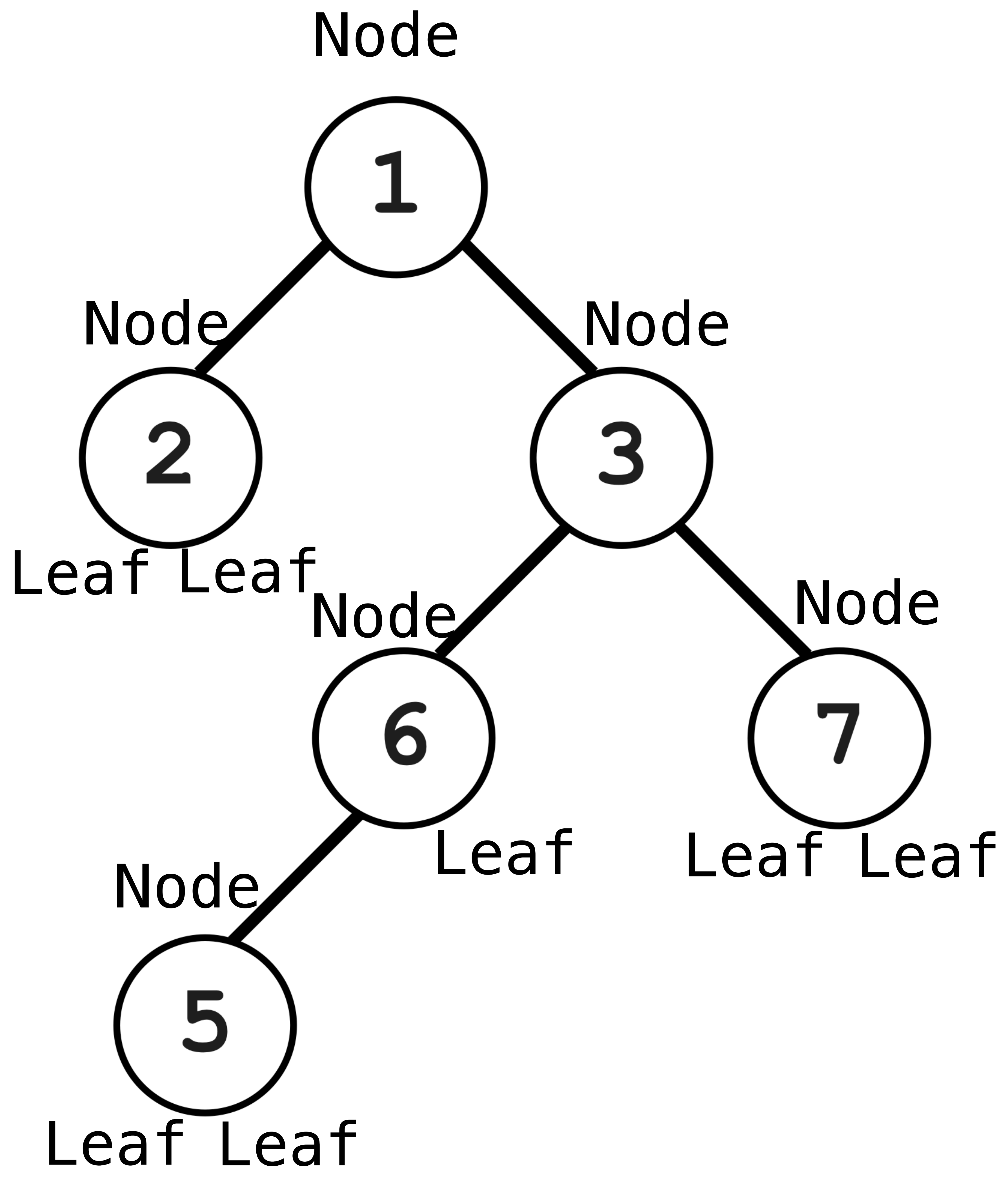
Error message

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

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

(tree examples: size/depth & traversals)

Lists with Multiple Element Types

Suppose we want to create a list that contains both integers and floats

» Define a function that counts the number of integers and floats; returns a tuple/record

» Define a function that sums up the elements of the list

» Define a function that converts the list into a list of floats

URM Programs

Recall a URM program:

» Z i: zero reg[i]

» I i: increment reg[i] register

» T I j: reg[j] <- reg[i]

» J I j k: jump to k-th instr. if reg[i] =reg[j]

Summary

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

Recursive and parametrized ADTs give us richer structure