Algebraic Data Types

Concepts of Programming Languages
Lecture 5

CAS CS 320

Practice Problem

Implement the function

```
val fill_in_steps : (int, int, int) -> int list
```

so that **fill_in_steps (a, b, c)** is the shortest list of numbers starting at **a**, ending at **c**, containing **b** and having consecutive adjacent elements

example: **fill_in_steps (1, 4, -2) = [1;2;3;4;3;2;1;0;-1;-2]**

Outline

- Discuss **tail-recursion**, and how it affects the design of functional programs
- Cover **algebraic data types (ADTs)** in more depth, including **recursive** ADTs and **parameterized** ADTs
- See some examples of useful/common ADTs

Learning Objectives

Write tail-recursive versions of functions

Determine whether or not simple programs are tail recursive

Work with and define recursive/parametrized ADTs (e.g., trees)

Tail Recursion

demo

(mod 2 the wrong way)

Tail Recursion

```
let rec fact n =
  if n <= 0
  then 1
  else n * fact (n - 1)
```

not tail recursive

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

tail recursive

A recursive function is **tail recursive** if it does not perform any computations on the result of a recursive call

Why do we care?

Why do we care?

Recursive functions are *expensive* with respect to the call-stack. We can't eliminate stack frames until *all* sub-calls finish

Why do we care?

Recursive functions are *expensive* with respect to the call-stack. We can't eliminate stack frames until *all* sub-calls finish

Tail-call elimination is an optimization implemented by OCaml's compiler which *reuses* stack frames, making recursive functions "behave iteratively" when executed

Why do we care?

Recursive functions are *expensive* with respect to the call-stack. We can't eliminate stack frames until *all* sub-calls finish

Tail-call elimination is an optimization implemented by OCaml's compiler which *reuses* stack frames, making recursive functions "behave iteratively" when executed

In Short: Tail-recursive functions are more memory efficient

The Picture

```
fact 5
```

```
let rec fact n =
  if n <= 0
  then 1
  else n * fact (n - 1)
```

The Picture

```
fact 5
```

```
fact 4
```

```
let rec fact n =
  if n <= 0
  then 1
  else n * fact (n - 1)
```

The Picture

```
let rec fact n =
  if n <= 0
  then 1
  else n * fact (n - 1)
```

```
fact 5
```

```
fact 4
```

```
fact 3
```

The Picture

```
let rec fact n =
  if n <= 0
  then 1
  else n * fact (n - 1)
```

```
fact 5
```

```
fact 4
```

```
fact 3
```

```
fact 2
```

The Picture

```
let rec fact n =
  if n <= 0
  then 1
  else n * fact (n - 1)
```

fact 5

fact 4

fact 3

fact 2

fact 1

The Picture

```
let rec fact n =
  if n <= 0
  then 1
  else n * fact (n - 1)
```

fact 5

fact 4

fact 3

fact 2

fact 1

fact 0

The Picture

```
let rec fact n =
  if n <= 0
  then 1
  else n * fact (n - 1)
```

fact 5

fact 4

fact 3

fact 2

fact 1

fact 0

$\Longrightarrow$ **1**

The Picture

```
let rec fact n =
  if n <= 0
  then 1
  else n * fact (n - 1)
```

```
fact 5

fact 4

fact 3

fact 2

fact 1
⟹ 1 * 1 = 1
```

The Picture

```
let rec fact n =
  if n <= 0
  then 1
  else n * fact (n - 1)
```

```
fact 5
```

```
fact 4
```

```
fact 3
```

```
fact 2

⟹ 2 * 1 = 2
```

The Picture

```
let rec fact n =
  if n <= 0
  then 1
  else n * fact (n - 1)
```

```
fact 5
```

```
fact 4
```

```
fact 3
⟹ 3 * 2 = 6
```

The Picture

```
fact 5
```

```
fact 4

⟹ 4 * 6 = 24
```

```
let rec fact n =
  if n <= 0
  then 1
  else n * fact (n - 1)
```

The Picture

```
fact 5
⟹ 5 * 24 = 120
```

```
let rec fact n =
  if n <= 0
  then 1
  else n * fact (n - 1)
```

The Picture

```
let rec fact n =
  if n <= 0
  then 1
  else n * fact (n - 1)
```

fact 5
$\Longrightarrow$ **5 * 24 = 120**

fact 4
$\Longrightarrow$ **4 * 6 = 24**

fact 3
$\Longrightarrow$ **3 * 2 = 6**

fact 2
$\Longrightarrow$ **2 * 1 = 2**

fact 1
$\Longrightarrow$ **1 * 1 = 1**

fact 0
$\Longrightarrow$ **1**

1 frame per recursive call

The Picture

```
loop 1 5
```

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

The Picture

```
loop 1 5
```

```
loop 5 4
```

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

The Picture

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

```
loop 1 5
```

```
loop 5 4
```

```
loop 20 3
```

The Picture

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

loop 1 5

loop 5 4

loop 20 3

fact 60 2

The Picture

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

loop 1 5
loop 5 4
loop 20 3
fact 60 2
fact 120 1

The Picture

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

```
loop 1 5
loop 5 4
loop 20 3
fact 60 2
fact 120 1
fact 120 0
```

The Picture

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

```
loop 1 5

loop 5 4

loop 20 3

fact 60 2

fact 120 1

fact 120 0
⟹ 120
```

The Picture

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

```
loop 1 5
```

```
loop 5 4
```

```
loop 20 3
```

```
fact 60 2
```

```
fact 120 1
⟹ 120
```

The Picture

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

```
loop 1 5
```

```
loop 5 4
```

```
loop 20 3
```

```
fact 60 2

⟹ 120
```

The Picture

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

```
loop 1 5
```

```
loop 5 4
```

```
loop 20 3

⟹ 120
```

The Picture

```
loop 1 5
```

```
loop 5 4
⟹ 120
```

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

The Picture

```
loop 1 5
⟹ 120
```

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

The Picture

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

```
loop 1 5
⟹ 120

loop 5 4
⟹ 120

loop 20 3
⟹ 120

fact 60 2
⟹ 120

fact 120 1
⟹ 120

fact 120 0
⟹ 120
```

1 frame per recursive call

BUT THE VALUE DOESN'T CHANGE ON IT'S WAY UP THE CALL STACK

The Picture (Optimized)

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

```
loop 1 5
```

The Picture (Optimized)

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

```
loop 5 4
```

The Picture (Optimized)

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

```
loop 20 3
```

The Picture (Optimized)

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

```
loop 120 1
```

The Picture (Optimized)

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

```
loop 120 0
```

The Picture (Optimized)

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

```
loop 120 0

⟹ 120
```

The Picture (Optimized)

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

```
loop 120 0

⟹ 120
```

1 frame
for every
recursive
call

Tail Position

```
let rec fact n =
  if n <= 0
  then 1
  else n * fact (n - 1)
```

computation after the recursive call

not tail recursive

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

tail recursive

Tail-call optimizations apply to functions whose recursive calls are in **tail position**

Intuition: A call is in tail position if there is no computation *after* the recursive call

Tail Position (A bit more Formally)

```
let rec f x1 x2 ... xk = e
```

Tail Position (A bit more Formally)

```
let rec f x1 x2 ... xk = e
```

A recursive call **f e1 e2 ... ek** is* in tail position in **e** if:

* f cannot appear in e1 ... ek

Tail Position (A bit more Formally)

```
let rec f x1 x2 ... xk = e
```

A recursive call **f e1 e2 ... ek** is* in tail position in **e** if:

- it does not appear in **e**, or **e** is the recursive call itself

* f cannot appear in e1 ... ek

Tail Position (A bit more Formally)

```
let rec f x1 x2 ... xk = e
```

A recursive call **f e1 e2 ... ek** is* in tail position in **e** if:

- it does not appear in **e**, or **e** is the recursive call itself
- e = **if e1 then e2 else e3** and the call does not appear in **e1** and it is in tail position in **e2** and **e3**

* f cannot appear in e1 ... ek

Tail Position (A bit more Formally)

```
let rec f x1 x2 ... xk = e
```

A recursive call **f e1 e2 ... ek** is* in tail position in **e** if:

- it does not appear in **e**, or **e** is the recursive call itself
- e = **if e1 then e2 else e3** and the call does not appear in **e1** and it is in tail position in **e2** and **e3**
- e is a **match-expression** and the call is in tail position in every branch, and does not appear in the matched expression

* f cannot appear in e1 ... ek

Tail Position (A bit more Formally)

```
let rec f x1 x2 ... xk = e
```

A recursive call **f e1 e2 ... ek** is* in tail position in **e** if:

- it does not appear in **e**, or **e** is the recursive call itself
- e = **if e1 then e2 else e3** and the call does not appear in **e1** and it is in tail position in **e2** and **e3**
- e is a **match-expression** and the call is in tail position in every branch, and does not appear in the matched expression
- e = **let x = e1 in e2** and the call does not appear in the **e1** and it is in tail position in **e2**

* f cannot appear in e1 ... ek

Tail Recursion and Lists

```
let append l r =
  let rec loop l acc =
    match l with
    | [] -> acc
    | x :: xs -> loop (x :: acc) xs
  in loop l r
```

Tail Recursion and Lists

```
let append l r =
  let rec loop l acc =
    match l with
    | [] -> acc
    | x :: xs -> loop (x :: acc) xs
  in loop l r
```

Take care with tail-recursion and lists

Tail Recursion and Lists

```
let append l r =
  let rec loop l acc =
    match l with
    | [] -> acc
    | x :: xs -> loop (x :: acc) xs
  in loop l r
```

Take care with tail-recursion and lists

Does the above program concatenate two lists?

Tail Recursion and Lists

```
let append l r =
  let rec loop l acc =
    match l with
    | [] -> acc
    | x :: xs -> loop (x :: acc) xs
  in loop l r
```

Take care with tail-recursion and lists

Does the above program concatenate two lists?

The Moral: Tail recursive functions on lists often reverse the lists

Tail Recursion and Lists

```
let append l r =
  let rec loop l acc =
    match l with
    | [] -> acc
    | x :: xs -> loop (x :: acc) xs
  in loop l r
```

should be (List.rev l)

Take care with tail-recursion and lists

Does the above program concatenate two lists?

The Moral: Tail recursive functions on lists often reverse the lists

Accumulators

```
let fact n =
  let rec loop acc n =
    if n <= 0
    then acc
    else loop (n * acc) (n - 1)
  in loop 1 n
```

Our accumulator pattern is almost always tail recursive

(though it's not the only way to write tail recursive functions)

Code Example

Implement the function

```
reverse : 'a list -> 'a list
```

in a tail-recursive fashion

Code Example

Implement the function which computes the nth Fibonacci number in a tail-recursive fashion

Algebraic Data Types

Recall: 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**

Recall: 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**

Recall: Pattern Matching

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

We work with variants (and any other type) by

» giving patterns a value can match with

» writing what to do in each case

Recall: Pattern Matching

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

We work with variants (and any other type) by

» giving patterns a value can match with

» writing what to do in each case

Recall: 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

Recall: 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

(Don't be fooled by the similarity in syntax)

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

(just things to keep in mind...)

Aside: Constructors are not function

```
type t = A of int function as an argument
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

(just things to keep in mind...)

Recursive ADTs

A Simple Observation

```
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**

Question. *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

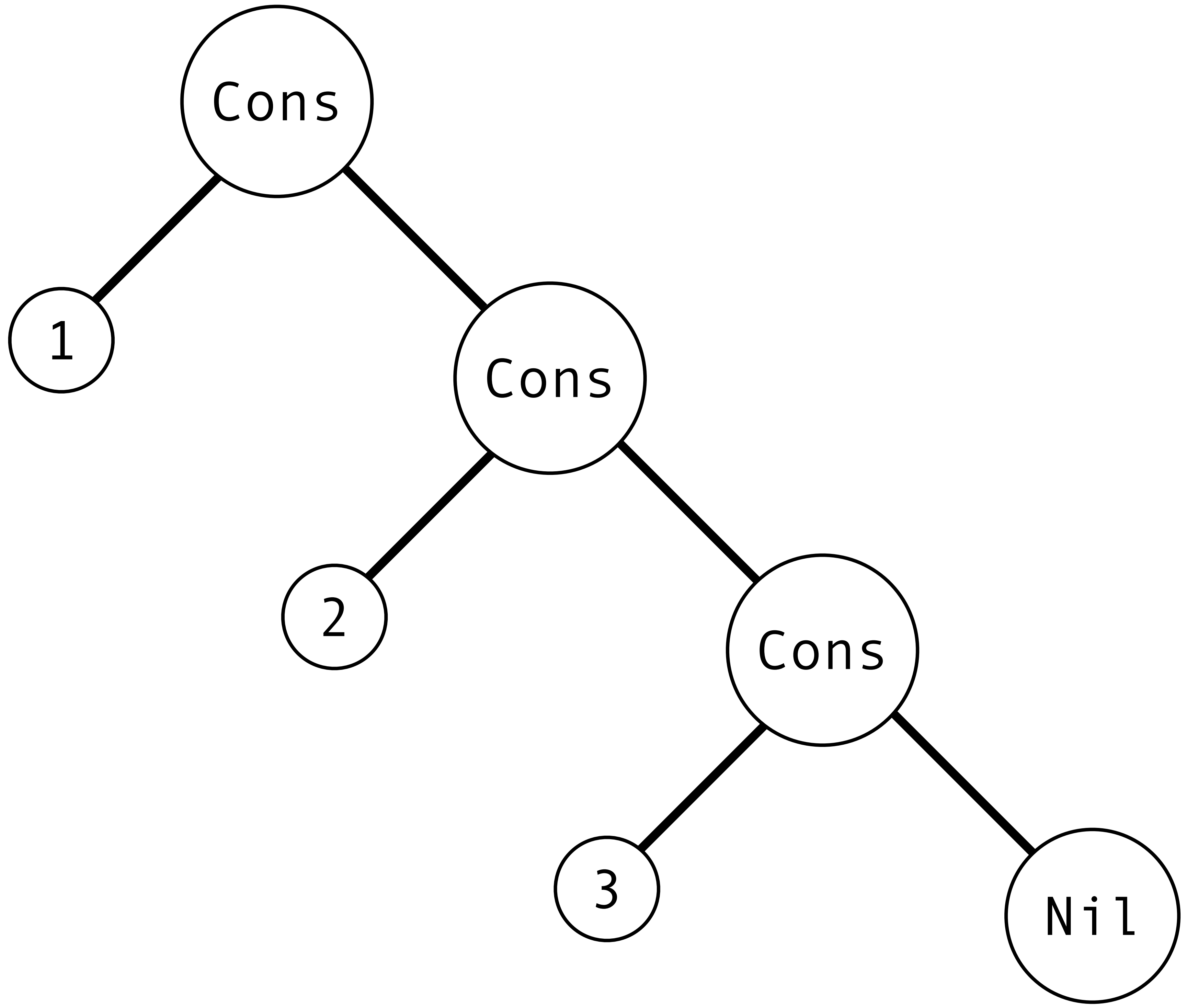
```
type intlist
  = Nil
  | Cons of int * intlist

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

Code Example

Implement the function

```
val snoc : intlist -> int -> intlist
```

so that ***snoc lst i*** *is the result of adding* ***i*** *to the* <u>*end*</u> *of* ***lst***

A More Interesting Example: Expressions

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

A More Interesting Example: Expressions

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

`Suppose we're building a calculator*`

`*This is exactly what we'll be doing when we build an interpreter.`

A More Interesting Example: Expressions

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

Suppose we're building a calculator*

Before we compute the value of an input, we first have to find an abstract representation of the input

*This is exactly what we'll be doing when we build an interpreter.

A More Interesting Example: Expressions

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

Suppose we're building a calculator*

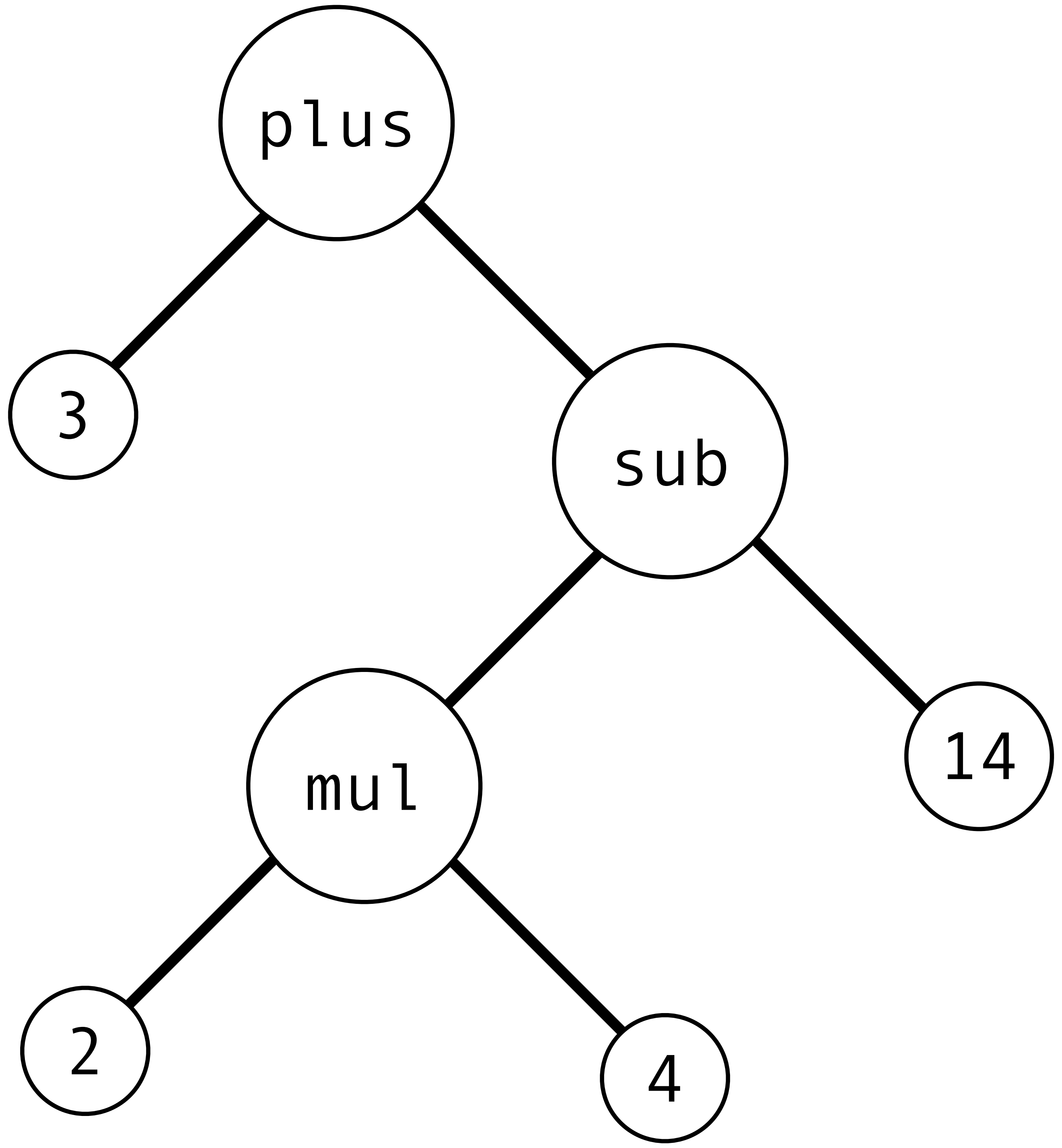
Before we compute the value of an input, we first have to find an abstract representation of the input

This will help us separate the tasks of evaluation and parsing

*This is exactly what we'll be doing when we build an interpreter.

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!

Code Example

Implement the function

```
val eval : expr -> int
```

so that ***eval exp*** *is the value of the given arithmetic expression*

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 type constructor
  = 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

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

No Ad-Hoc Polymorphism

```
let add (a : int) (b : int) : int = a + b
let add (a : string) (b : string) : string = a ^ b (* This overwrite above *)
let add (a : 'a list) (b : 'a list) : 'a list = a @ b (* This overwrites above *)
```

No Ad-Hoc Polymorphism

```
let add (a : int) (b : int) : int = a + b
let add (a : string) (b : string) : string = a ^ b (* This overwrite above *)
let add (a : 'a list) (b : 'a list) : 'a list = a @ b (* This overwrites above *)
```

There is no overloading in OCaml

No Ad-Hoc Polymorphism

```
let add (a : int) (b : int) : int = a + b
let add (a : string) (b : string) : string = a ^ b (* This overwrite above *)
let add (a : 'a list) (b : 'a list) : 'a list = a @ b (* This overwrites above *)
```

There is no overloading in OCaml

"Parametric" here means we must be type agnostic:

No Ad-Hoc Polymorphism

```
let add (a : int) (b : int) : int = a + b
let add (a : string) (b : string) : string = a ^ b (* This overwrite above *)
let add (a : 'a list) (b : 'a list) : 'a list = a @ b (* This overwrites above *)
```

There is no overloading in OCaml

"Parametric" here means we **must** be type agnostic:

» It has to work for *all* types

No Ad-Hoc Polymorphism

```
let add (a : int) (b : int) : int = a + b
let add (a : string) (b : string) : string = a ^ b (* This overwrite above *)
let add (a : 'a list) (b : 'a list) : 'a list = a @ b (* This overwrites above *)
```

There is no overloading in OCaml

"Parametric" here means we **must** be type agnostic:

» It has to work for *all* types

» We can't do different computations for different types

Useful ADTs

Options

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

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

Options

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

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

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

Options

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

let head (l : 'a list) : 'a myoption =
  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. [27]

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)

(it's a bit of a buzz-term, bit we accept it)

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

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
    ...
```

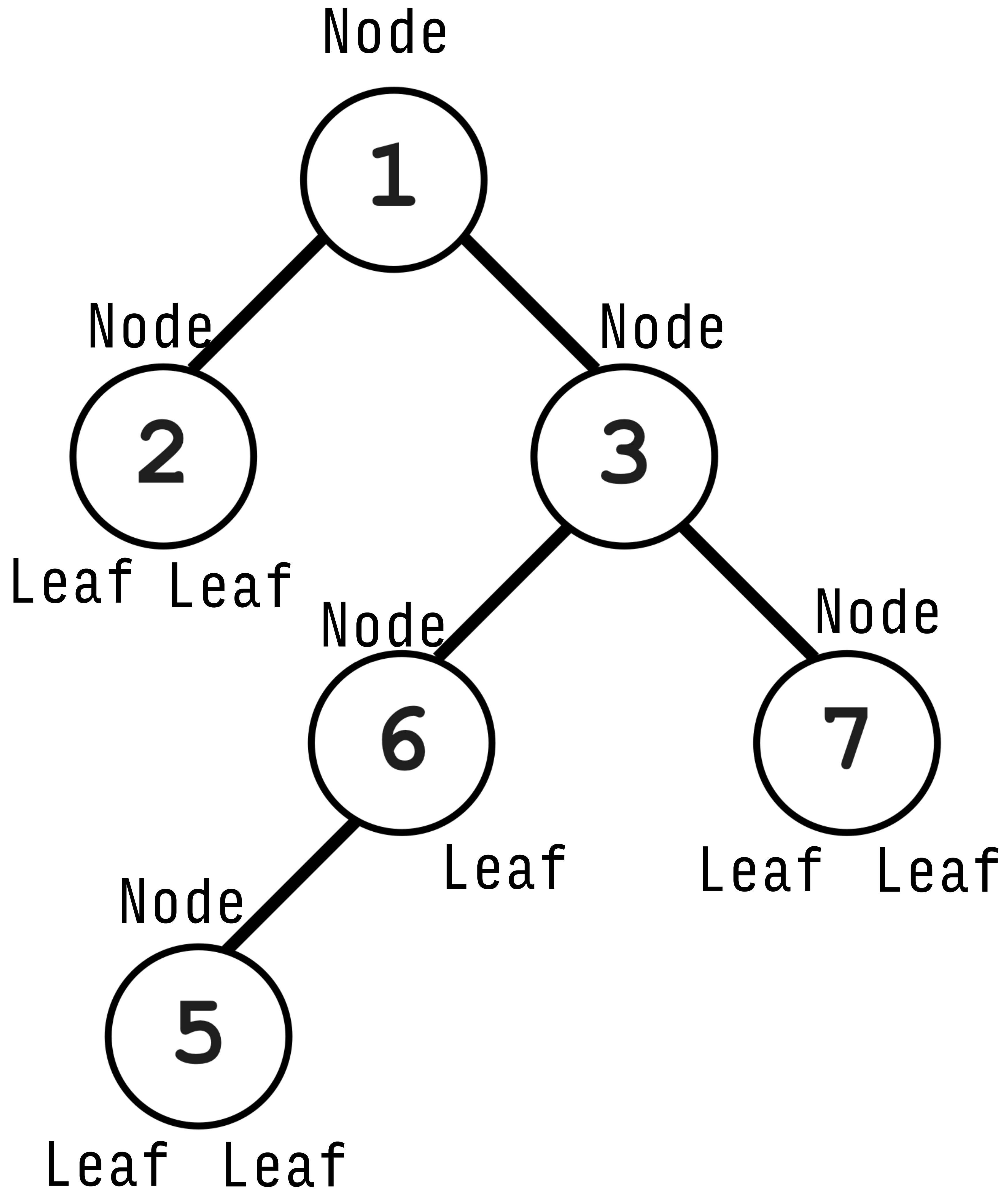
lists and optionals and results are built into OCaml.

You can also use the **#show** directive 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



Code Example

Implement

```
val size : 'a tree -> int
```

which determines the number of elements in the tree

I'll leave it as an exercise to implement the usual interface for trees in OCaml

Summary

- » ADTs help us organize data and create abstract interfaces
- » Recursive and parametrized ADTs give is richer structure
- » Tail-recursive functions help us write memory efficient functional code