Inductive Types I: Introduction

Principles of Programming Languages
Lecture 6

CAS CS 320

Introduction

Administrivia

Assignment 2 is due by 11:59PM on Thursday.

REPL is an REU about programming languages.

OCaml won the 2023 ACM PL Software award.

Objectives

Take a deeper dive into algebraic data types (ADTs).

Look primarily at recursive and parametrized algebraic ADTs.

Practice with some examples.

Keywords

`algebraic data types`

`simple variants`

`pattern matching`

`constructors`

`data-carrying variants`

`recursive variants`

`parametric variants`

`lists, options, results`

Practice Problem

Implement the function ***downup*** *which given*

n : nonnegative integer

returns the list of integers from n down to 1 then back up to n.

Your implementation must be linear time. As a challenge, try to write it tail-recursively.

Algebraic Data Types

Recall: Simple Variants

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

Recall: Simple Variants

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

Simple variants are like Java **Enums**.

Recall: Simple Variants

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

Simple variants are like Java **Enums**.

They are used to create small collections of different values.

Recall: Simple Variants

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

Simple variants are like Java **Enums**.

They are used to create small collections of different values.

Example. Above is a simple variant for different operating systems.

Recall: Simple Variants

constructor

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

Simple variants are like Java **Enums**.

They are used to create small collections of different values.

Example. Above is a simple variant for different operating systems.

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
```

variable pattern

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

variable pattern

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

An Aside: Constructor Arguments are not Tuples

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

An 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.

An 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.

An 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.)

An 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 *)
```

An 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.

An 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.

An 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...)

An Aside: Constructors are not function

function as an argument

```
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...)

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.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.

Understanding Check

```
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.*

Recursive ADTs

A Simple Observation

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

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

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`**`.`

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?

Simple Example: Lists

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

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

Simple 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.

Simple 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.

Reminder: Pattern Matching

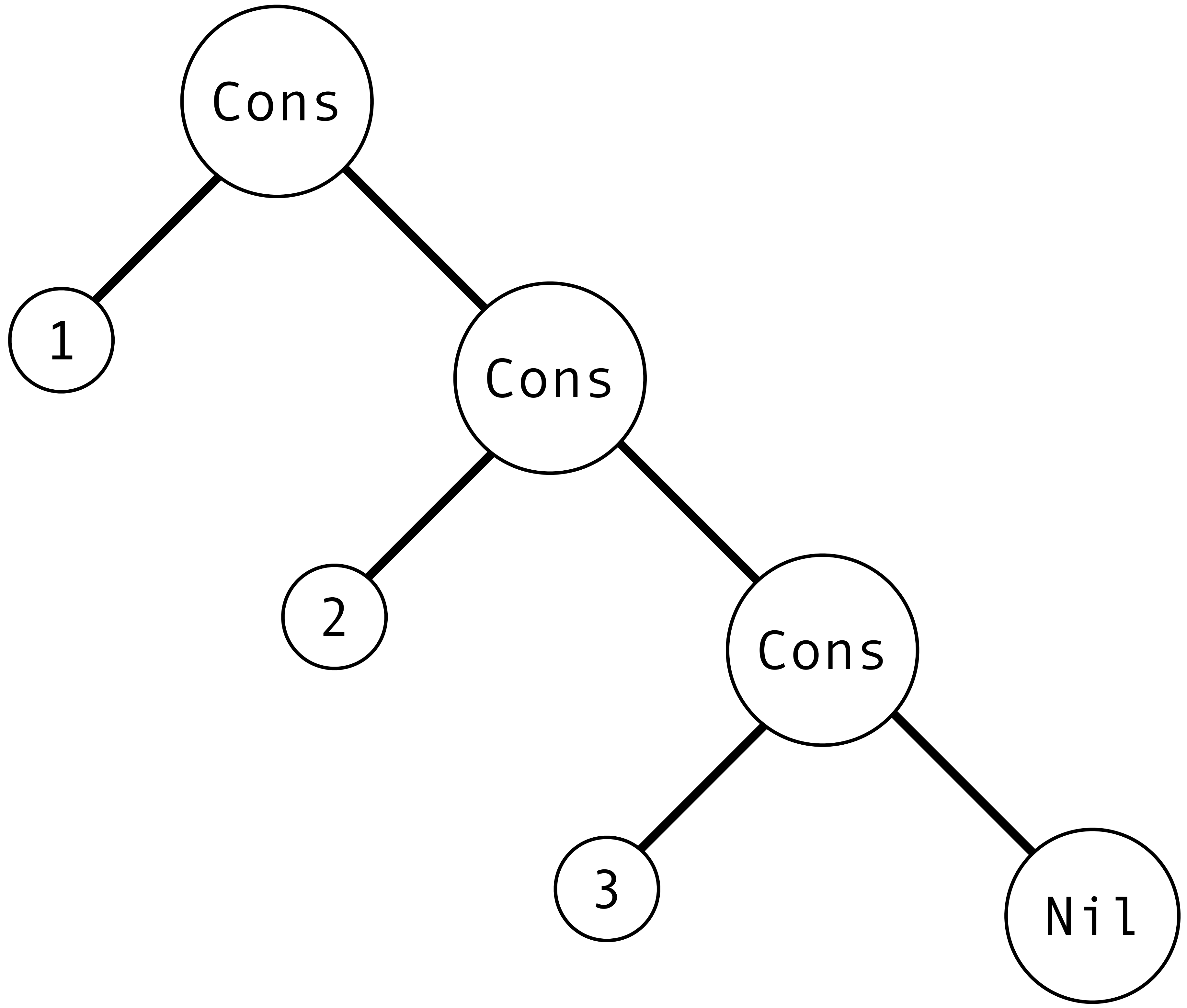
```
let rec snoc (xs : intlist) (x : int) : intlist =
  match xs with
  | Nil -> Cons (x, Nil)
  | Cons (y, ys) -> Cons (y, snoc ys x)

let _ = assert
  (snoc example 4 = Cons (1, Cons (2, Cons (3, Cons (4, Nil)))))
```

When we pattern match on a variant, the constructors are possible patterns.

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.*`

`*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.*

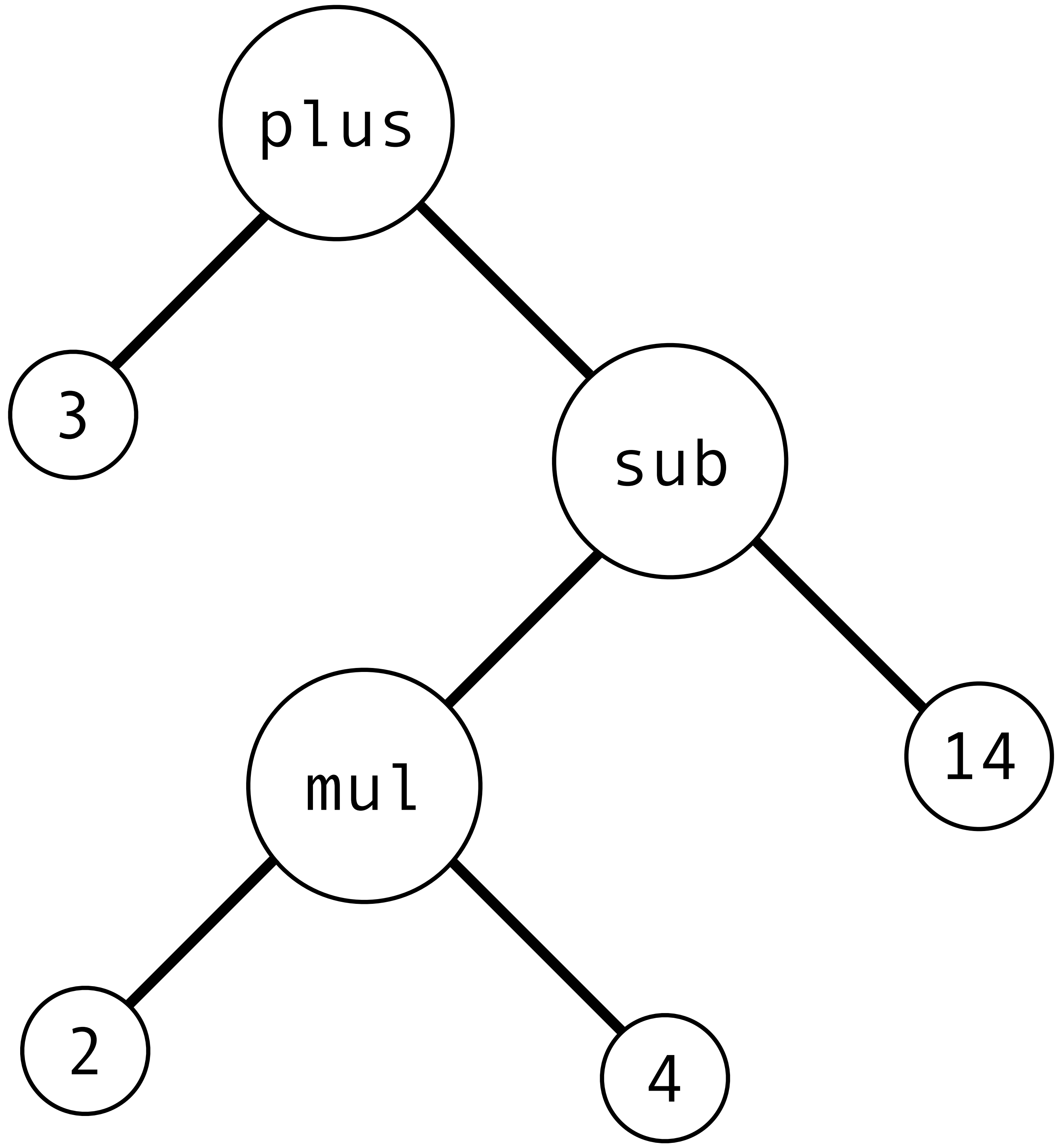
Before we compute the value of an input, we first have to find an abstract representation of the input.

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

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



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

A More Interesting Example: Expressions

```ocaml
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 in OCaml as a recursive variant.

Understanding Check

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

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

Write a function ***eval*** *of type* ***expr -> int****, which given an expression, determines its value.*

For example, ***eval x*** *should be* ***–3****.*

An Aside: Grammars

Let's take a look at OCaml's grammar for <u>expressions</u>.

Expressions look a lot like recursive variants...

Recursive records

```
type node = {
  head : int ;
  tail : node ;
}
```

Recursive records

```
type node = {
  head : int ;
  tail : node ;
}
```

As you might expect, records can also be recursive.

Recursive records

```
type node = {
  head : int ;
  tail : node ;
}
```

As you might expect, records can also be recursive.

But this is not a terribly useful structure...

(Why?)

Mutually Recursive Structures

```
type node = {head : int ; tail : maybetail}
and  maybetail = Nope | Yep of node
```

We can make it more useful by using mutual recursion.

The tail is now optional.

A Note on Evaluation

```
let not_good =
  let rec go i = i :: go (i + 1) in go 0
```

A Note on Evaluation

```
let not_good =
  let rec go i = i :: go (i + 1) in go 0
```

This type-checks, but causes a stack-overflow.

A Note on Evaluation

```
let not_good =
  let rec go i = i :: go (i + 1) in go 0
```

This type-checks, but causes a stack-overflow.

OCaml evaluates expressions according to the **call-by-value** so expressions are completely computed before being:

- bound to a name
- passed to a function

An Aside: What about this?

```
let rec what_about_this =
  1 :: 2 :: what_about_this
```

An Aside: What about this?

```
let rec what_about_this =
  1 :: 2 :: what_about_this
```

This is actually okay (but strange).

An Aside: What about this?

```
let rec what_about_this =
  1 :: 2 :: what_about_this
```

This is actually okay (but strange).

It's the cyclic list with **[1; 2; 1; 2;...]**.

An Aside: What about this?

```
let rec what_about_this =
  1 :: 2 :: what_about_this
```

This is actually okay (but strange).

It's the cyclic list with **[1; 2; 1; 2;...]**.

It's a recursive definition of a value, there is no computation, so there is no stack overflow.

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 *)
```

Note. There is no function 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 *)
```

Note. There is no function 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 *)
```

Note. There is no function 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.

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.

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.

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
    ...
```

<u>list</u>s and <u>optional</u>s and <u>result</u>s are built into OCaml.

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

Understanding Check

*Implement the function **first_three** which given*

l : a' list

*returns a **result** with a 3-element tuple in the case that l has at least 3 elements, and an **string** error message otherwise.*

Summary

Variants can be data-carrying, recursive and parametric.

This is all we need to be able to do most interesting things in OCaml.