Last time: monads (etc.)

This time: generic programming

```
val show : 'a → string
```

Generic functions

Data types

unit

```
type unit =
 Unit : unit
```

booleans

```
type bool =
   False: bool
 | True : bool
```

natural numbers

```
type nat =
    Zero: nat
  | Succ: nat → nat
```

sums

```
type ('a,'b) sum =
  Left : 'a → ('a,'b) sum
| Right: 'b → ('a,'b) sum
```

pairs

```
type ('a,'b) pair =
 Pair: 'a * 'b → ('a,'b) pair
```

lists

```
type 'a list =
  Nil : 'a list
| Cons: 'a * 'a list → 'a list
```

Data type operations: formatting

unit

```
let string_of_unit : unit → string = function
  Unit → "Unit"
```

booleans

```
let string_of_bool : bool → string = function
    False → "False"
  | True  → "True"
```

natural numbers

```
let rec string_of_nat : nat → string = function
    Zero   → "Zero"
  | Succ n → "(Succ "^ string_of_nat n ^")"
```

Data type operations: formatting (continued)

sums

```
let string_of_sum :
 ('a → string) → ('b → string) → ('a,'b) sum → string =
 fun l r → function
   Left x → "(Left "^ l x ^")"
 | Right y → "(Right "^ r y ^")"
```

pairs

```
let string_of_pair :
 ('a → string) → ('b → string) → ('a,'b) pair → string =
 fun l r → function
   Pair (x, y) → "(Pair "^ l x ^","^ r y ^")"
```

lists

```
let rec string_of_list :
 ('a → string) → 'a list → string =
 fun a → function
   Nil → "Nil"
 | Cons (x,xs) → "(Cons "^ a x ^","^ string_of_list a y ^")"
```

Operations defined on (most) data

equality

`'a → 'a → bool`

hashing

`'a → int`

ordering

`'a → 'a → int`

mapping

`('b → 'b) → 'a → 'a`

querying

`('b → bool) → 'a → 'b list`

pretty-printing

`'a → string`

parsing

`string → 'a`

serialising

`'a → string`

sizing

`'a → int`

Generic functions and parametricity

Some built-in OCaml functions are incompatible with parametricity:

```
val (=) : 'a → 'a → bool

val hash : 'a → int

val from_string : string → int → 'a
```

Generic functions and parametricity

Some built-in OCaml functions are incompatible with parametricity:

```
val (=) : 'a → 'a → bool

val hash : 'a → int

val from_string : string → int → 'a
```

How might we do better? Pass a description of the data **shape**:

```
val (=) : {D:DATA} → D.t → D.t → bool

val hash : {D:DATA} → D.t → int

val from_string : {D:DATA} → string → int → D.t
```

Data shape descriptions: type-indexed values

Idea: give an instance of some signature `T` for each OCaml types:

```
module T_int : T with type t = int
module T_bool : T with type t = bool
module T_pair (A:T) (B:T) : T with type t = A.t * B.t
module T_list (A:T) : T with type t = A.t list
module T_option (A:T) : T with type t = A.t option
(* etc. *)
```

`int` is represented by a module

```
T_int : T with type t = int
```

`int * bool` is represented by a module

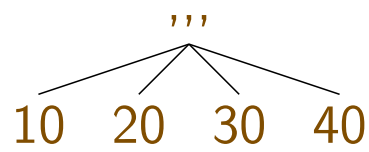
```
T_pair(T_int)(T_bool): T with type t = int * bool
```

`int option list` is represented by a module

```
T_list(T_option(T_int)): T with type t = int option list
```

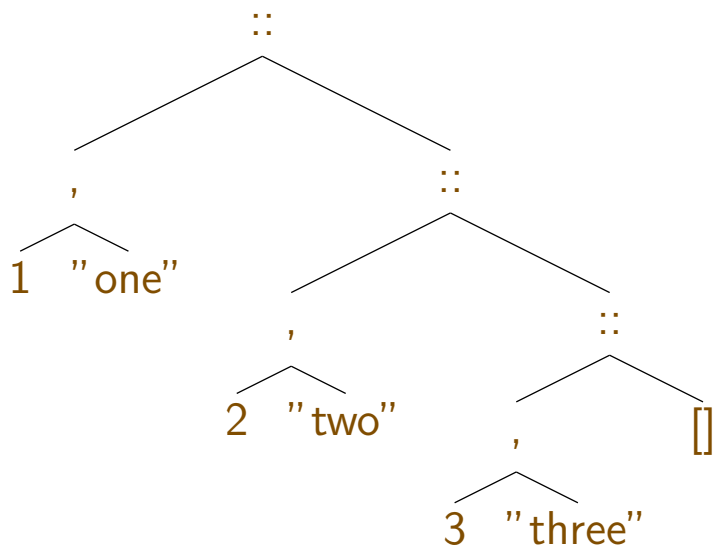
(etc.)

Data as trees




```
L ()
```

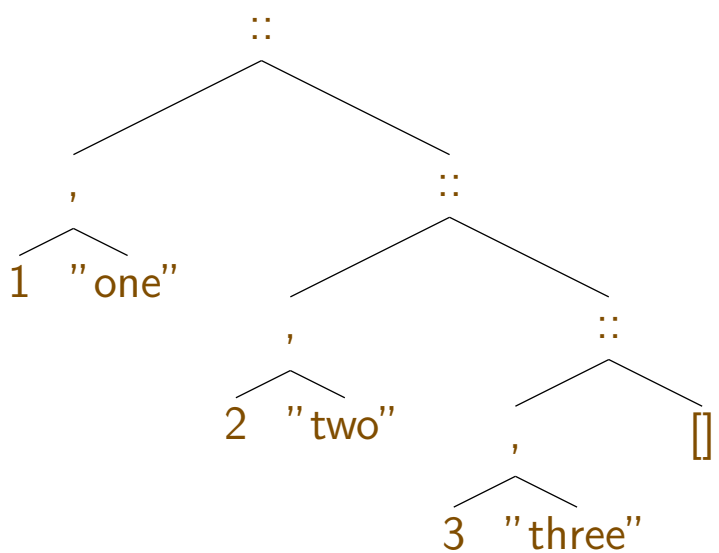
```
(10,20,30,40)
```




```
[(1, "one"); (2, "two"); (3, "three")]
```

Generic operations: three questions about data

1. **What type is this data?**
2. **What are its subnodes?**
3. **What about the recursive case?**



1. Examining types

Determining type equality

A type **representation**:

```
type _ type_rep
```

A type equality **test** that returns a proof on success:

```
val eqrep : 'a type_rep → 'b type_rep → ('a,'b) eql option
```

```
# eqrep ty_int ty_float
- : (int, float) eql option = None

# eqrep ty_int ty_int
- : (int, int) eql option = Some Refl
```

Type indexed values for type equality

```
module type TYPEABLE = sig
  type t
  val type_rep : t type_rep
  val eqrep : 'other type_rep → (t, 'other) eql option
end
```

```
implicit module Typeable_int : TYPEABLE with type t = int
implicit module Typeable_bool : TYPEABLE with type t = bool
implicit module Typeable_list{A:TYPEABLE}
(* etc. *)
```

Representing types

How should we define `type_rep`?

```
type _ type_rep =
  Int : int type_rep
| Bool : bool type_rep
| List : 'a type_rep → 'a list type_rep
| Option : 'a type_rep → 'a option type_rep
| Pair : 'a type_rep * 'b type_rep → ('a * 'b) type_rep
```

Implementing type equality

```
let rec eqrep :
 type a b.a typerep → b typerep → (a,b) eql option =
 fun l r → match l, r with
    Int, Int → Some Refl
  | Bool, Bool → Some Refl
  | List s, List t →
     (match eqrep s t with
        Some Refl → Some Refl
      | None → None)
  | Option s, Option t →
     (match eqrep s t with
        Some Refl → Some Refl
      | None → None)
  | Pair (s1, s2), Pair (t1, t2) →
     (match eqrep s1 t1, eqrep s2 t2 with
        Some Refl, Some Refl → Some Refl
      | _ → None)
  | _ → None
```

Implementing type equality

```
let rec eqrep :
 type a b.a typerep → b typerep → (a,b) eql option =
 fun l r → match l, r with
    Int, Int → Some Refl
  | Bool, Bool → Some Refl
  | List s, List t →
     (match eqrep s t with
        Some Refl → Some Refl
      | None → None)
  | Option s, Option t →
     (match eqrep s t with
        Some Refl → Some Refl
      | None → None)
  | Pair (s1, s2), Pair (t1, t2) →
     (match eqrep s1 t1, eqrep s2 t2 with
        Some Refl, Some Refl → Some Refl
      | _ → None)
  | _ → None
```

Problem: this representation has no support for user-defined types.

Extensible variants

Defining

```
type 'a t = ..
```

Extending

```
type 'a t +=
  A : int list → int t
| B : float list → 'a t
```

Constructing

```
A [1;2;3]  (* No different to standard variants *)
```

Matching

```
let f : type a. a t → string = function
  A _ → "A"
| B _ → "B"
| _ → "unknown" (* All matches must be open *)
```

Representing types, extensibly

```
type _ type_rep = ..

type _ type_rep += List : 'a type_rep → 'a list type_rep

implicit module Typeable_list{A:TYPEABLE}
  : TYPEABLE with type t = A.t list =
struct
  type t = A.t list
  let type_rep = List A.type_rep
  let eqrep : type b.b type_rep → (A.t list,b) eql option =
   function
    | List b → (match A.eqrep b with
                  Some Refl → Some Refl
                | None → None)
    | _ → None
end
```

Implementing type equality, extensibly

```
module type TYPEABLE = sig
  type t
  val type_rep : t type_rep
  val eqrep : 'other type_rep → (t, 'other) eql option
end
```

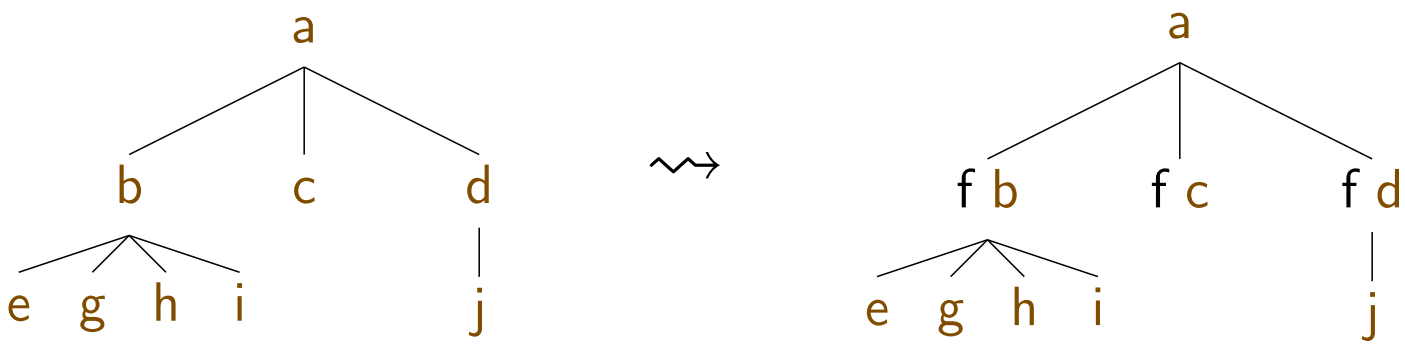
```
val eqty : {A:TYPEABLE} → {B:TYPEABLE} →
   (A.t, B.t) eq option

let eqty {A:TYPEABLE} {B:TYPEABLE} = A.eqrep B.type_rep
```

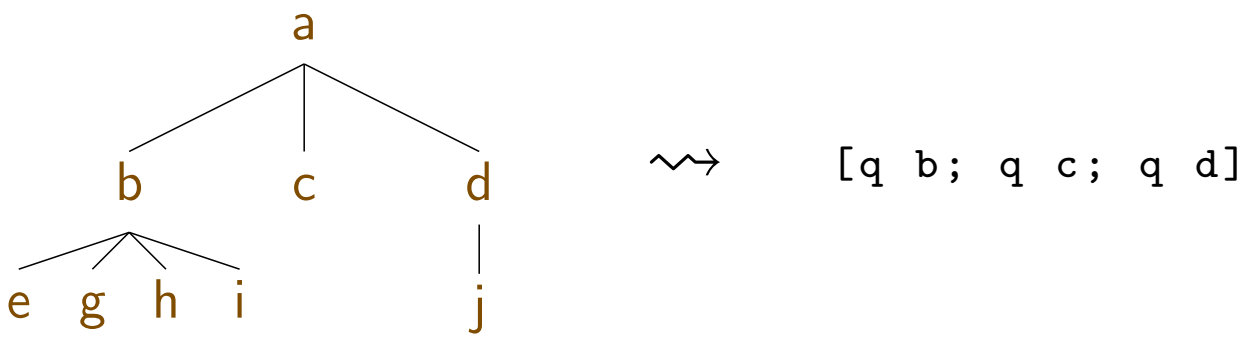
2. Accessing subnodes

Traversing datatypes

gmapT



gmapQ



An interface for accessing subnodes

```
module type DATA
```

```
type genericT = {D:DATA} → D.t → D.t
type 'u genericQ = {D:DATA} → D.t → 'u

val gmapT : genericT → genericT
val gmapQ : 'u genericQ → 'u list genericQ
```

A signature for accessing subnodes

```
module type DATA = sig
  type t
  module Typeable : TYPEABLE with type t = t
  val gmapT : genericT → t → t
  val gmapQ : 'u genericQ → t → 'u list
end
```

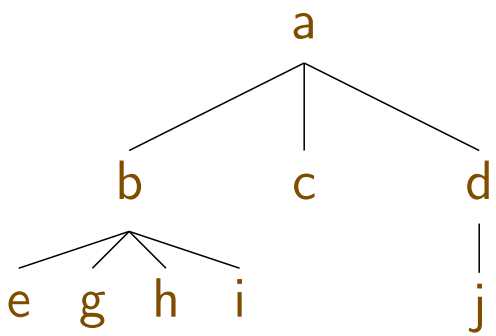
```
implicit module Data_int : DATA with type t = int
implicit module Data_list{A:DATA} : DATA with type t = A.t list
(etc.)
```

Polymorphic types for generic traversals: gmapT


```
type genericT = {D:DATA} → D.t → D.t

val gmapT : genericT → genericT
```

Polymorphic types for generic queries: gmapQ



$\rightsquigarrow$ `[q b; q c; q d]`

```
type 'u genericQ = {D:DATA} → D.t → 'u

val gmapQ : 'u genericQ → 'u list genericQ
```

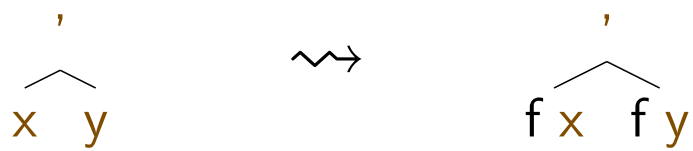
Traversing datatypes: primitive types

$$x \rightsquigarrow x$$

```
gmapT {Data_int} f
```

```
implicit module Data_int : DATA with type t = int =
struct
  type t = int
  module Typeable = Typeable_int
  let gmapT f x = x
  let gmapQ f x = []
end
```

Traversing datatypes: pairs

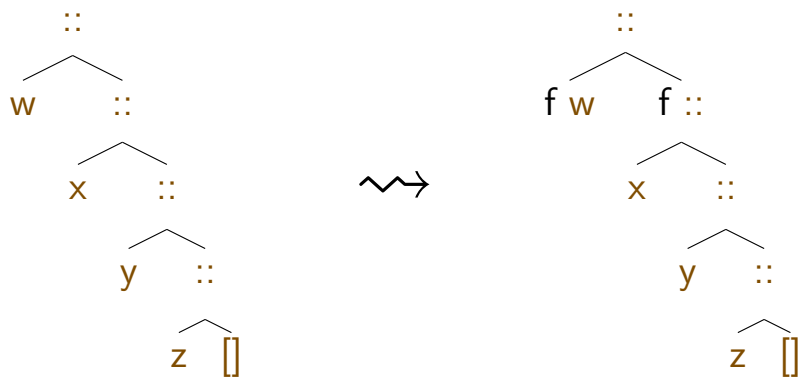



```
gmapT {Data_pair{A}{B}} f
```

With DATA instances A and B for the type parameters:

```
implicit module Data_pair {A: DATA} {B: DATA}
 : DATA with type t = A.t * B.t =
struct
  type t = A.t * B.t
  module Typeable = Typeable_pair{A.Typeable}{B.Typeable}
  let gmapT f (x, y) = (f x, f y)
  let gmapQ q (x, y) = [q x; q y]
end
```

Traversing datatypes: lists



`gmapT (list a)f`

```
implicit module rec Data_list {A: DATA} :
   DATA with type t = A.t list = struct
  type t = A.t list
  module Typeable = Typeable_list{A.Typeable}
  let gmapT f l =
    match l with [] → [] | x :: xs → f x :: f xs
  let gmapQ q l =
    match l with [] → [] | x :: xs → [q x; q xs]
end
```

(Disclaimer: `implicit module rec` not yet supported)

3. Handling recursion

Generic maps, bottom up

```
let rec everywhere : genericT → genericT =
  fun f {X:DATA} x → f (gmapT (everywhere f) x)
```

In practice a few more annotations are needed:

```
let rec everywhere : genericT → genericT =
 fun (f : genericT) {X:DATA} x →
    f ((gmapT (everywhere f) : genericT) x)
```

Generic maps, top down

```
let rec everywhere' : genericT → genericT =
  fun f {X:DATA} x → gmapT (everywhere' f) (f x)
```

With annotations:

```
let rec everywhere' : genericT → genericT =
  fun (f : genericT) {X:DATA} x →
     (gmapT (everywhere' f) : genericT) (f {X} x)
```

Generic maps with a stop condition

```
val everywhereBut : bool genericQ → genericT → genericT
```

```
let rec everywhereBut :
    bool genericQ → genericT → genericT =
  fun stop f {X:DATA} x →
    if stop x then x
    else f (gmapT (everywhereBut stop f) x)
```

Using generic maps

```
val everywhere : genericT → genericT
```

```
let mkT : {T:TYPEABLE} → (T.t → T.t) → genericT =
  fun {T:TYPEABLE} g {D: DATA} x →
    match eqty {T} {D:TYPEABLE} with
      | Some Refl → g x
      | None → x
```

(In practice, we need an auxiliary function: see the notebook.)

```
everywhere (mkT succ) [(false, 1); (false, 2); (true, 3)]
```

Generic queries

```
let rec everything : 'r. ('r → 'r → 'r) → 'r genericQ → '
    r genericQ =
  fun join g {X: DATA} x →
    fold_left join (g x)
      (gmapQ (everything join g) x)
```

```
let rec everythingBut :
  'r. ('r → 'r → 'r) → ('r * bool) genericQ → 'r genericQ =
  fun join stop {X: DATA} x →
    match stop x with
    | v, true → v
    | v, false → fold_left join v
                   (gmapQ (everythingBut join stop) x)
```

Using generic queries

```
val everything :
  ('r → 'r → 'r) → 'r genericQ → 'r genericQ =
```

```
let mkQ : 'u. {T:TYPEABLE} → 'u → (T.t → 'u) → 'u genericQ =
  fun {T:TYPEABLE} u g {D: DATA} x →
   match eqty {T} {D.Typeable} with
       | Some Refl → g x
       | None → u
```

```
everything (@) (mkQ [] (fun x →  [x])) [(1,false); (2,true)]
```

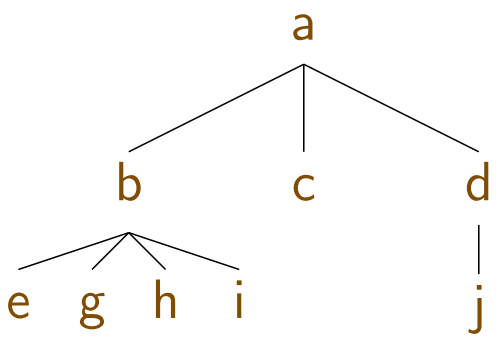
Generic printing

Representing constructors

Add an additional field to `data` for distinguishing constructors:

```
module type DATA = sig
  type t
  module Typeable : TYPEABLE with type t = t
  val gmapT : genericT → t → t
  val gmapQ : 'u genericQ → t → 'u list
  val constructor_: t → string
end
```

A generic printing function

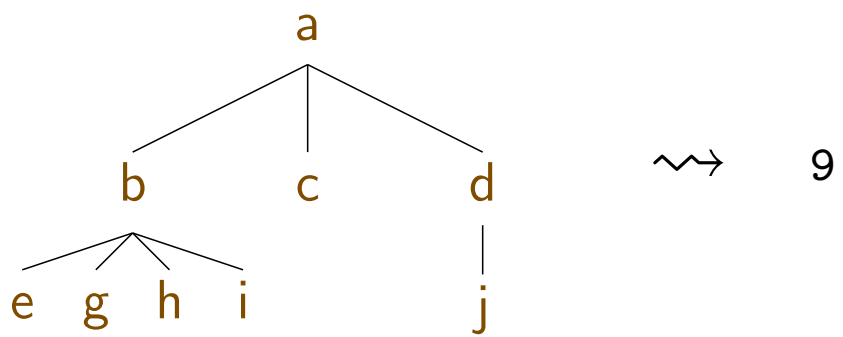


$\rightsquigarrow$ `"(a (b (e) (g) (h) (i)) (c) (d (j)))"`

```
let rec gshow {D:DATA} (v : D.t) =
  "("^ constructor_ v ^ String.concat " " (gmapQ gshow v) ^ ")"
```

Generic sizing

Computing value size generically




```
let sum l = List.fold_left (+) 0 l

let rec gsize {D:DATA} (v : D.t) = 1 + sum (gmapQ gsize v)


gsize 3

gsize [1;2;3]

gsize [(1,false); (2,false); (3,true)]
```

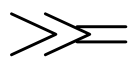
Main remaining problem: performance

Generic traversals are slow!

Solution: **staging** (next week)

`.< e >.`

Next time: monads (etc.), continued



```
effect E
```