Camllib 1.2.0

February 1, 2011

Contents

Chapter 1

Module Print : Printing functions using module Format

1.1 Printing functions for standard datatypes (lists,arrays,...)

In the following functions, optional arguments `?first`, `?sep`, `?last` denotes the formatting instructions (under the form of a `format` string) issued at the beginning, between two elements, and at the end. The functional argument(s) indicate(s) how to print elements.

```
val list :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  (Format.formatter -> 'a -> unit) -> Format.formatter -> 'a list -> unit
```

Print a list

```
val array :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  (Format.formatter -> 'a -> unit) -> Format.formatter -> 'a array -> unit
```

Print an array

```
val pair :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  (Format.formatter -> 'a -> unit) ->
  (Format.formatter -> 'b -> unit) -> Format.formatter -> 'a * 'b -> unit
```

Print a pair

```
val option :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  (Format.formatter -> 'a -> unit) -> Format.formatter -> 'a option -> unit
```

Print an optional element

```
val hash :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  ?firstbind:(unit, Format.formatter, unit) Pervasives.format ->
  ?sepbind:(unit, Format.formatter, unit) Pervasives.format ->
  ?lastbind:(unit, Format.formatter, unit) Pervasives.format ->
  (Format.formatter -> 'a -> unit) ->
  (Format.formatter -> 'b -> unit) ->
  Format.formatter -> ('a, 'b) Hashtbl.t -> unit
```

Print an hashtable

```
val weak :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  (Format.formatter -> 'a -> unit) -> Format.formatter -> 'a Weak.t -> unit
```

Print a weak pointer array

1.2 Useful functions

```
val string_of_print : (Format.formatter -> 'a -> unit) -> 'a -> string
```

Transforms a printing function into a conversion-to-string function.

```
val print_of_string : ('a -> string) -> Format.formatter -> 'a -> unit
```

Transforms a conversion-to-string function to a printing function.

```
val sprintf :
  ?margin:int -> ('a, Format.formatter, unit, string) Pervasives.format4 -> 'a
```

Better `sprintf` function than `Format.sprintf`, as it takes the same kind of formatters as other `Format.Xprintf` functions.

```
val escaped : ?linebreak:char -> string -> string
```

Escape a string, replacing line breaks by `linebreak` (default `'\n'`). When used for DOT output, `'\l'` and `'\r'` produces respectively left or right justified lines, instead of center justified lines.

Chapter 2

Module `Sette` : Sets over ordered types (extension of standard library module and polymorphic variant)

This module implements the set data structure, given a total ordering function over the set elements. All operations over sets are purely applicative (no side-effects). The implementation uses balanced binary trees, and is therefore reasonably efficient: insertion and membership take time logarithmic in the size of the set, for instance.

Modified by B. Jeannet to get a generic type and a few additions (like conversions form and to maps and pretty-printing).

```
type 'a set =
  | Empty
  | Node of 'a set * 'a * 'a set * int
```

Meant to be internal, but exporting needed for Mappe.maptoset.

```
type 'a t = 'a set
```

The type of sets over elements of type 'a.

```
val empty : 'a t
```

The empty set.

```
val is_empty : 'a t -> bool
```

Test whether a set is empty or not.

```
val mem : 'a -> 'a t -> bool
```

`mem x s` tests whether `x` belongs to the set `s`.

```
val add : 'a -> 'a t -> 'a t
```

`add x s` returns a set containing all elements of `s`, plus `x`. If `x` was already in `s`, `s` is returned unchanged.

```
val singleton : 'a -> 'a t
```

`singleton x` returns the one-element set containing only `x`.

```
val remove : 'a -> 'a t -> 'a t
```

`remove x s` returns a set containing all elements of `s`, except `x`. If `x` was not in `s`, `s` is returned unchanged.

`val union : 'a t -> 'a t -> 'a t`

`val inter : 'a t -> 'a t -> 'a t`

`val diff : 'a t -> 'a t -> 'a t`

Union, intersection and set difference.

`val compare : 'a t -> 'a t -> int`

Total ordering between sets. Can be used as the ordering function for doing sets of sets.

`val equal : 'a t -> 'a t -> bool`

`equal s1 s2` tests whether the sets `s1` and `s2` are equal, that is, contain equal elements.

`val subset : 'a t -> 'a t -> bool`

`subset s1 s2` tests whether the set `s1` is a subset of the set `s2`.

`val iter : ('a -> unit) -> 'a t -> unit`

`iter f s` applies `f` in turn to all elements of `s`. The order in which the elements of `s` are presented to `f` is unspecified.

`val fold : ('a -> 'b -> 'b) -> 'a t -> 'b -> 'b`

`fold f s a` computes `(f xN ... (f x2 (f x1 a))...)`, where `x1 ... xN` are the elements of `s`. The order in which elements of `s` are presented to `f` is unspecified.

Raises `Not_found` if no fount

Returns the computed accumulator

`val for_all : ('a -> bool) -> 'a t -> bool`

`for_all p s` checks if all elements of the set satisfy the predicate `p`.

`val exists : ('a -> bool) -> 'a t -> bool`

`exists p s` checks if at least one element of the set satisfies the predicate `p`.

`val filter : ('a -> bool) -> 'a t -> 'a t`

`filter p s` returns the set of all elements in `s` that satisfy predicate `p`.

`val partition : ('a -> bool) -> 'a t -> 'a t * 'a t`

`partition p s` returns a pair of sets `(s1, s2)`, where `s1` is the set of all the elements of `s` that satisfy the predicate `p`, and `s2` is the set of all the elements of `s` that do not satisfy `p`.

`val cardinal : 'a t -> int`

Return the number of elements of a set.

`val elements : 'a t -> 'a list`

Return the list of all elements of the given set. The returned list is sorted in increasing order with respect to the ordering `Pervasives.compare`.

`val min_elt : 'a t -> 'a`

Return the smallest element of the given set (with respect to the `Ord.compare` ordering), or raise `Not_found` if the set is empty.

`val max_elt : 'a t -> 'a`

Same as `min_elt`, but returns the largest element of the given set.

`val choose : 'a t -> 'a`

Return one element of the given set, or raise `Not_found` if the set is empty. Which element is chosen is unspecified, but equal elements will be chosen for equal sets.

```
val print :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  (Format.formatter -> 'a -> unit) -> Format.formatter -> 'a t -> unit
```

```
module type S =
  sig
```

```
type elt
```

The type of the set elements.

```
type t
```

The type of sets.

```
val repr : t -> elt Sette.set
val obj : elt Sette.set -> t
module Ord :
Set.OrderedType  with type t=elt
```

The ordering module used for this set module.

```
val empty : t
```

The empty set.

```
val is_empty : t -> bool
```

Test whether a set is empty or not.

```
val mem : elt -> t -> bool
```

`mem x s` tests whether `x` belongs to the set `s`.

```
val add : elt -> t -> t
```

`add x s` returns a set containing all elements of `s`, plus `x`. If `x` was already in `s`, `s` is returned unchanged.

```
val singleton : elt -> t
```

`singleton x` returns the one-element set containing only `x`.

```
val remove : elt -> t -> t
```

`remove x s` returns a set containing all elements of `s`, except `x`. If `x` was not in `s`, `s` is returned unchanged.

```
val union : t -> t -> t
```

Set union.

```
val inter : t -> t -> t
```

Set intersection.

`val diff : t -> t -> t`

Set difference.

`val compare : t -> t -> int`

Total ordering between sets. Can be used as the ordering function for doing sets of sets.

`val equal : t -> t -> bool`

`equal s1 s2` tests whether the sets `s1` and `s2` are equal, that is, contain equal elements.

`val subset : t -> t -> bool`

`subset s1 s2` tests whether the set `s1` is a subset of the set `s2`.

`val iter : (elt -> unit) -> t -> unit`

`iter f s` applies `f` in turn to all elements of `s`. The order in which the elements of `s` are presented to `f` is unspecified.

`val fold : (elt -> 'a -> 'a) -> t -> 'a -> 'a`

`fold f s a` computes `(f xN ... (f x2 (f x1 a))...)`, where `x1 ... xN` are the elements of `s`. The order in which elements of `s` are presented to `f` is unspecified.

`val for_all : (elt -> bool) -> t -> bool`

`for_all p s` checks if all elements of the set satisfy the predicate `p`.

`val exists : (elt -> bool) -> t -> bool`

`exists p s` checks if at least one element of the set satisfies the predicate `p`.

`val filter : (elt -> bool) -> t -> t`

`filter p s` returns the set of all elements in `s` that satisfy predicate `p`.

`val partition : (elt -> bool) -> t -> t * t`

`partition p s` returns a pair of sets `(s1, s2)`, where `s1` is the set of all the elements of `s` that satisfy the predicate `p`, and `s2` is the set of all the elements of `s` that do not satisfy `p`.

`val cardinal : t -> int`

Return the number of elements of a set.

`val elements : t -> elt list`

Return the list of all elements of the given set. The returned list is sorted in increasing order with respect to the ordering `Ord.compare`, where `Ord` is the argument given to `Sette.Make`[2].

`val min_elt : t -> elt`

Return the smallest element of the given set (with respect to the `Ord.compare` ordering), or raise `Not_found` if the set is empty.

`val max_elt : t -> elt`

Same as `Sette.S.min_elt`[2], but returns the largest element of the given set.

`val choose : t -> elt`

Return one element of the given set, or raise `Not_found` if the set is empty. Which element is chosen is unspecified, but equal elements will be chosen for equal sets.

```
    val print :
      ?first:(unit, Format.formatter, unit) Pervasives.format ->
      ?sep:(unit, Format.formatter, unit) Pervasives.format ->
      ?last:(unit, Format.formatter, unit) Pervasives.format ->
      (Format.formatter -> elt -> unit) ->
      Format.formatter -> t -> unit
  end
```

Output signature of the functor `Sette.Make`[2].

```
module Make :
functor (Ord :  Set.OrderedType) -> S  with type elt = Ord.t  and module Ord=Ord
```

Functor building an implementation of the set structure given a totally ordered type.

```
module Compare :
  sig

    val split :
      ('a -> 'a -> int) -> 'a -> 'a Sette.t -> 'a Sette.t * bool * 'a Sette.t
```

Meant to be internal, but exporting needed for Mappe.maptoset.

```
    val add : ('a -> 'a -> int) -> 'a -> 'a Sette.t -> 'a Sette.t
    val mem : ('a -> 'a -> int) -> 'a -> 'a Sette.t -> bool
    val remove : ('a -> 'a -> int) -> 'a -> 'a Sette.t -> 'a Sette.t
    val union : ('a -> 'a -> int) -> 'a Sette.t -> 'a Sette.t -> 'a Sette.t
    val inter : ('a -> 'a -> int) -> 'a Sette.t -> 'a Sette.t -> 'a Sette.t
    val diff : ('a -> 'a -> int) -> 'a Sette.t -> 'a Sette.t -> 'a Sette.t
    val equal : ('a -> 'a -> int) -> 'a Sette.t -> 'a Sette.t -> bool
    val compare : ('a -> 'a -> int) -> 'a Sette.t -> 'a Sette.t -> int
    val subset : ('a -> 'a -> int) -> 'a Sette.t -> 'a Sette.t -> bool
    val filter : ('a -> 'a -> int) -> ('a -> bool) -> 'a Sette.t -> 'a Sette.t
    val partition :
      ('a -> 'a -> int) -> ('a -> bool) -> 'a Sette.t -> 'a Sette.t * 'a Sette.t
  end
```

Chapter 3

Module PSette : Sets over ordered types, parametrized polymorphic version

Same interface as Sette[2], but each set stores its comparison function.

If compiled without -noassert option, checks for binary operations that the two arguments refers to the same comparison function (by testing physical equality).

BE CAUTIOUS: do not use Pervasives.compare function directly, as it is an external function, so that writing t1.cmp <- Pervasives.compare induces the creation of a closure, and if later t2.cmp <- Pervasives.compare is executed, t1.cmp != t2.cmp because two different closures have been created. Instead, first define let compare = fun x y -> Pervasives.compare x y in ....

```
type 'a t = {
  compare : 'a -> 'a -> int ;
  set : 'a Sette.t ;
}
```

```
val empty : ('a -> 'a -> int) -> 'a t
```

The empty set.

```
val is_empty : 'a t -> bool
```

Test whether a set is empty or not.

```
val mem : 'a -> 'a t -> bool
```

mem x s tests whether x belongs to the set s.

```
val add : 'a -> 'a t -> 'a t
```

add x s returns a set containing all elements of s, plus x. If x was already in s, s is returned unchanged.

```
val singleton : ('a -> 'a -> int) -> 'a -> 'a t
```

singleton x returns the one-element set containing only x.

```
val remove : 'a -> 'a t -> 'a t
```

remove x s returns a set containing all elements of s, except x. If x was not in s, s is returned unchanged.

```
val union : 'a t -> 'a t -> 'a t
val inter : 'a t -> 'a t -> 'a t
val diff : 'a t -> 'a t -> 'a t
```

Union, intersection and set difference.

```
val compare : 'a t -> 'a t -> int
```

Total ordering between sets. Can be used as the ordering function for doing sets of sets.

```
val equal : 'a t -> 'a t -> bool
```

`equal s1 s2` tests whether the sets `s1` and `s2` are equal, that is, contain equal elements.

```
val subset : 'a t -> 'a t -> bool
```

`subset s1 s2` tests whether the set `s1` is a subset of the set `s2`.

```
val iter : ('a -> unit) -> 'a t -> unit
```

`iter f s` applies `f` in turn to all elements of `s`. The order in which the elements of `s` are presented to `f` is unspecified.

```
val fold : ('a -> 'b -> 'b) -> 'a t -> 'b -> 'b
```

`fold f s a` computes `(f xN ... (f x2 (f x1 a))...)`, where `x1 ... xN` are the elements of `s`. The order in which elements of `s` are presented to `f` is unspecified.

Raises `Not_found` if no fount

Returns the computed accumulator

```
val for_all : ('a -> bool) -> 'a t -> bool
```

`for_all p s` checks if all elements of the set satisfy the predicate `p`.

```
val exists : ('a -> bool) -> 'a t -> bool
```

`exists p s` checks if at least one element of the set satisfies the predicate `p`.

```
val filter : ('a -> bool) -> 'a t -> 'a t
```

`filter p s` returns the set of all elements in `s` that satisfy predicate `p`.

```
val partition : ('a -> bool) -> 'a t -> 'a t * 'a t
```

`partition p s` returns a pair of sets `(s1, s2)`, where `s1` is the set of all the elements of `s` that satisfy the predicate `p`, and `s2` is the set of all the elements of `s` that do not satisfy `p`.

```
val cardinal : 'a t -> int
```

Return the number of elements of a set.

```
val elements : 'a t -> 'a list
```

Return the list of all elements of the given set. The returned list is sorted in increasing order with respect to the ordering `Pervasives.compare`.

```
val min_elt : 'a t -> 'a
```

Return the smallest element of the given set (with respect to the `Ord.compare` ordering), or raise `Not_found` if the set is empty.

```
val max_elt : 'a t -> 'a
```

Same as `min_elt`, but returns the largest element of the given set.

```
val choose : 'a t -> 'a
```

Return one element of the given set, or raise `Not_found` if the set is empty. Which element is chosen is unspecified, but equal elements will be chosen for equal sets.

```
val print :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  (Format.formatter -> 'a -> unit) -> Format.formatter -> 'a t -> unit
```

```
val make : ('a -> 'a -> int) -> 'a Sette.t -> 'a t
```

Internal, do not use

Chapter 4

Module `Mappe` : Association tables over ordered types (extension of standard library module and polymorphic variant)

This module implements applicative association tables, also known as finite maps or dictionaries, given a total ordering function over the keys. All operations over maps are purely applicative (no side-effects). The implementation uses balanced binary trees, and therefore searching and insertion take time logarithmic in the size of the map.

Modification par B. Jeannet pour avoir des mappes génériques

```
type ('a, 'b) map =
  | Empty
  | Node of ('a, 'b) map * 'a * 'b * ('a, 'b) map * int
```

```
type ('a, 'b) t = ('a, 'b) map
```

The type of maps from type `'a` to type `'b`.

```
val is_empty : ('a, 'b) t -> bool
```

Is the map empty ?

```
val empty : ('a, 'b) t
```

The empty map.

```
val add : 'a -> 'b -> ('a, 'b) t -> ('a, 'b) t
```

`add x y m` returns a map containing the same bindings as `m`, plus a binding of `x` to `y`. If `x` was already bound in `m`, its previous binding disappears.

```
val find : 'a -> ('a, 'b) t -> 'b
```

`find x m` returns the current binding of `x` in `m`, or raises `Not_found` if no such binding exists.

```
val remove : 'a -> ('a, 'b) t -> ('a, 'b) t
```

`remove x m` returns a map containing the same bindings as `m`, except for `x` which is unbound in the returned map.

```
val mem : 'a -> ('a, 'b) t -> bool
```

`mem x m` returns `true` if `m` contains a binding for `m`, and `false` otherwise.

```
val addmap : ('a, 'b) t -> ('a, 'b) t -> ('a, 'b) t
```

`addmap m1 m2` merges the two maps `m1` and `m2`. If a key of `m2` was already bound in `m1`, the previous binding in `a` disappears.

```
val merge : ('a -> 'a -> 'a) -> ('b, 'a) t -> ('b, 'a) t -> ('b, 'a) t
```

`merge mergedata a b` is similar to `addmap a b`, but if a key `k` is bound to `d1` in `m1` and to `d2` in `m2`, the key `k` is bound to `mergedata d1 d2` in the result

```
val mergei :
  ('a -> 'b -> 'b -> 'b) ->
  ('a, 'b) t -> ('a, 'b) t -> ('a, 'b) t
```

Same as `merge`, but the function receives as arguments the key (of the first map)

```
val common : ('a -> 'b -> 'c) -> ('d, 'a) t -> ('d, 'b) t -> ('d, 'c) t
```

`common commondata a b` returns a map the keys of which must be keys in both `a` and in `b`, and those keys are bound to `interdata d1 d2`.

```
val commoni :
  ('a -> 'b -> 'c -> 'd) ->
  ('a, 'b) t -> ('a, 'c) t -> ('a, 'd) t
```

Same as `common`, but the function receives as arguments the key (of the first map)

```
val combine :
  ('a -> 'b option -> 'c option -> 'd option) ->
  ('a, 'b) t -> ('a, 'c) t -> ('a, 'd) t
val interset : ('a, 'b) t -> 'a Sette.t -> ('a, 'b) t
```

`interset map set` selects the bindings in `a` whose key belongs to `set`.

```
val diffset : ('a, 'b) t -> 'a Sette.t -> ('a, 'b) t
```

`interset map set` removes the bindings of `a` whose key belongs to `set`.

```
val iter : ('a -> 'b -> unit) -> ('a, 'b) t -> unit
```

`iter f m` applies `f` to all bindings in map `m`. `f` receives the key as first argument, and the associated value as second argument. The order in which the bindings are passed to `f` is unspecified. Only current bindings are presented to `f`: bindings hidden by more recent bindings are not passed to `f`.

```
val map : ('a -> 'b) -> ('c, 'a) t -> ('c, 'b) t
```

`map f m` returns a map with same domain as `m`, where the associated value `a` of all bindings of `m` has been replaced by the result of the application of `f` to `a`. The order in which the associated values are passed to `f` is unspecified.

```
val mapi : ('a -> 'b -> 'c) -> ('a, 'b) t -> ('a, 'c) t
```

Same as `map`, but the function receives as arguments both the key and the associated value for each binding of the map.

```
val fold : ('a -> 'b -> 'c -> 'c) -> ('a, 'b) t -> 'c -> 'c
```

`fold f m a` computes `(f kN dN ... (f k1 d1 a)...)`, where `k1 ... kN` are the keys of all bindings in `m`, and `d1 ... dN` are the associated data. The order in which the bindings are presented to `f` is unspecified.

```
val maptoset : ('a, 'b) t -> 'a Sette.t
```

`maptoset m` returns the set of the keys in the association table

```
val mapofset : ('a -> 'b) -> 'a Sette.t -> ('a, 'b) t
```

`mapofset f s` returns the map associating `f key` to `key`, for each element `key` of the set `s`

```
val compare : ('a -> 'b -> int) -> ('c, 'a) t -> ('c, 'b) t -> int
val comparei : ('a -> 'b -> 'c -> int) -> ('a, 'b) t -> ('a, 'c) t -> int
```

Comparison function between maps, total if the comparison function for data is total

```
val equal : ('a -> 'b -> bool) -> ('c, 'a) t -> ('c, 'b) t -> bool
val equali : ('a -> 'b -> 'c -> bool) -> ('a, 'b) t -> ('a, 'c) t -> bool
```

equality between maps

```
val subset : ('a -> 'b -> bool) -> ('c, 'a) t -> ('c, 'b) t -> bool
val subseti : ('a -> 'b -> 'c -> bool) -> ('a, 'b) t -> ('a, 'c) t -> bool
```

subset between maps

```
val filter : ('a -> 'b -> bool) -> ('a, 'b) t -> ('a, 'b) t
```

`filter p m` returns the map of all bindings in `m` that satisfy predicate `p`.

```
val partition : ('a -> 'b -> bool) -> ('a, 'b) t -> ('a, 'b) t * ('a, 'b) t
```

`partition p m` returns a pair of maps `(m1, m2)`, where `m1` is the map of all the bindings of `m` that satisfy the predicate `p`, and `m2` is the map of all the bindings of `m` that do not satisfy `p`.

```
val cardinal : ('a, 'b) t -> int
```

Number of keys of a map

```
val bindings : ('a, 'b) t -> ('a * 'b) list
```

Return the list of all bindings of the given map. The returned list is sorted in increasing order with respect to the ordering `Pervasives.compare` on keys.

```
val min_key : ('a, 'b) t -> 'a
```

Return the smallest key of the given map or raise `Not_found` if the set is empty.

```
val max_key : ('a, 'b) t -> 'a
```

Same as `min_elt`, but returns the largest key of the given map.

```
val choose : ('a, 'b) t -> 'a * 'b
```

Returns a binding, or raise `Not_found` if empty

```
val print :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  ?firstbind:(unit, Format.formatter, unit) Pervasives.format ->
  ?sepbind:(unit, Format.formatter, unit) Pervasives.format ->
  ?lastbind:(unit, Format.formatter, unit) Pervasives.format ->
  (Format.formatter -> 'a -> unit) ->
  (Format.formatter -> 'b -> unit) ->
  Format.formatter -> ('a, 'b) t -> unit
```

Print a map.

```
module type S =
  sig

     type key
     type 'a t
     module Setkey :
     Sette.S  with type elt=key
     val repr : 'a t -> (key, 'a) Mappe.map
     val obj : (key, 'a) Mappe.map -> 'a t
     val is_empty : 'a t -> bool
     val empty : 'a t
     val add : key -> 'a -> 'a t -> 'a t
     val find : key -> 'a t -> 'a
     val remove : key -> 'a t -> 'a t
     val mem : key -> 'a t -> bool
     val addmap : 'a t -> 'a t -> 'a t
     val merge : ('a -> 'a -> 'a) -> 'a t -> 'a t -> 'a t
     val mergei : (key -> 'a -> 'a -> 'a) ->
       'a t -> 'a t -> 'a t
     val common : ('a -> 'b -> 'c) -> 'a t -> 'b t -> 'c t
     val commoni : (key -> 'a -> 'b -> 'c) ->
       'a t -> 'b t -> 'c t
     val combine :
       (key -> 'a option -> 'b option -> 'c option) ->
       'a t -> 'b t -> 'c t
     val interset : 'a t -> Setkey.t -> 'a t
     val diffset : 'a t -> Setkey.t -> 'a t
     val iter : (key -> 'a -> unit) -> 'a t -> unit
     val map : ('a -> 'b) -> 'a t -> 'b t
     val mapi : (key -> 'a -> 'b) -> 'a t -> 'b t
     val fold : (key -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b
     val maptoset : 'a t -> Setkey.t
     val mapofset : (key -> 'a) -> Setkey.t -> 'a t
     val compare : ('a -> 'b -> int) -> 'a t -> 'b t -> int
     val comparei : (key -> 'a -> 'b -> int) -> 'a t -> 'b t -> int
     val equal : ('a -> 'b -> bool) -> 'a t -> 'b t -> bool
     val equali : (key -> 'a -> 'b -> bool) -> 'a t -> 'b t -> bool
     val subset : ('a -> 'b -> bool) -> 'a t -> 'b t -> bool
     val subseti : (key -> 'a -> 'b -> bool) -> 'a t -> 'b t -> bool
     val filter : (key -> 'a -> bool) -> 'a t -> 'a t
     val partition : (key -> 'a -> bool) -> 'a t -> 'a t * 'a t
     val cardinal : 'a t -> int
     val bindings : 'a t -> (key * 'a) list
     val min_key : 'a t -> key
     val max_key : 'a t -> key
     val choose : 'a t -> key * 'a
```

```
    val print :
      ?first:(unit, Format.formatter, unit) Pervasives.format ->
      ?sep:(unit, Format.formatter, unit) Pervasives.format ->
      ?last:(unit, Format.formatter, unit) Pervasives.format ->
      ?firstbind:(unit, Format.formatter, unit) Pervasives.format ->
      ?sepbind:(unit, Format.formatter, unit) Pervasives.format ->
      ?lastbind:(unit, Format.formatter, unit) Pervasives.format ->
      (Format.formatter -> key -> unit) ->
      (Format.formatter -> 'a -> unit) -> Format.formatter -> 'a t -> unit
  end
```

Output signature of the functor `Mappe.Make`[4].

```
module Make :
functor (Setkey :  Sette.S) -> S  with type key=Setkey.elt and module Setkey=Setkey
```

Functor building an implementation of the map structure given a totally ordered type.

```
module Compare :
  sig

    val add :
      ('a -> 'a -> int) -> 'a -> 'b -> ('a, 'b) Mappe.t -> ('a, 'b) Mappe.t
    val find : ('a -> 'a -> int) -> 'a -> ('a, 'b) Mappe.t -> 'b
    val remove : ('a -> 'a -> int) -> 'a -> ('a, 'b) Mappe.t -> ('a, 'b) Mappe.t
    val mem : ('a -> 'a -> int) -> 'a -> ('a, 'b) Mappe.t -> bool
    val addmap :
      ('a -> 'a -> int) -> ('a, 'b) Mappe.t -> ('a, 'b) Mappe.t -> ('a, 'b) Mappe.t
    val merge :
      ('a -> 'a -> int) ->
      ('b -> 'b -> 'b) -> ('a, 'b) Mappe.t -> ('a, 'b) Mappe.t -> ('a, 'b) Mappe.t
    val mergei :
      ('a -> 'a -> int) ->
      ('a -> 'b -> 'b -> 'b) ->
      ('a, 'b) Mappe.t -> ('a, 'b) Mappe.t -> ('a, 'b) Mappe.t
    val common :
      ('a -> 'a -> int) ->
      ('b -> 'c -> 'd) -> ('a, 'b) Mappe.t -> ('a, 'c) Mappe.t -> ('a, 'd) Mappe.t
    val commoni :
      ('a -> 'a -> int) ->
      ('a -> 'b -> 'c -> 'd) ->
      ('a, 'b) Mappe.t -> ('a, 'c) Mappe.t -> ('a, 'd) Mappe.t
    val combine :
      ('a -> 'a -> int) ->
      ('a -> 'b option -> 'c option -> 'd option) ->
      ('a, 'b) Mappe.t -> ('a, 'c) Mappe.t -> ('a, 'd) Mappe.t
    val interset :
      ('a -> 'a -> int) -> ('a, 'b) Mappe.t -> 'a Sette.t -> ('a, 'b) Mappe.t
    val diffset :
      ('a -> 'a -> int) -> ('a, 'b) Mappe.t -> 'a Sette.t -> ('a, 'b) Mappe.t
    val compare :
      ('a -> 'a -> int) ->
      ('b -> 'c -> int) -> ('a, 'b) Mappe.t -> ('a, 'c) Mappe.t -> int
    val comparei :
```

```
    ('a -> 'a -> int) ->
    ('a -> 'b -> 'c -> int) -> ('a, 'b) Mappe.t -> ('a, 'c) Mappe.t -> int
  val equal :
    ('a -> 'a -> int) ->
    ('b -> 'c -> bool) -> ('a, 'b) Mappe.t -> ('a, 'c) Mappe.t -> bool
  val equali :
    ('a -> 'a -> int) ->
    ('a -> 'b -> 'c -> bool) -> ('a, 'b) Mappe.t -> ('a, 'c) Mappe.t -> bool
  val subset :
    ('a -> 'a -> int) ->
    ('b -> 'c -> bool) -> ('a, 'b) Mappe.t -> ('a, 'c) Mappe.t -> bool
  val subseti :
    ('a -> 'a -> int) ->
    ('a -> 'b -> 'c -> bool) -> ('a, 'b) Mappe.t -> ('a, 'c) Mappe.t -> bool
  val filter :
    ('a -> 'a -> int) ->
    ('a -> 'b -> bool) -> ('a, 'b) Mappe.t -> ('a, 'b) Mappe.t
  val partition :
    ('a -> 'a -> int) ->
    ('a -> 'b -> bool) -> ('a, 'b) Mappe.t -> ('a, 'b) Mappe.t * ('a, 'b) Mappe.t
end
```

Chapter 5

Module PMappe : Association tables over ordered types, parametrized polymorphic version

Same interface as Mappe[4], but each map stores its comparison function.

If compiled without -noassert option, checks for binary operations that the two arguments refers to the same comparison function (by testing physical equality).

BE CAUTIOUS: do not use Pervasives.compare function directly, as it is an external function, so that writing t1.cmp <- Pervasives.compare induces the creation of a closure, and if later t2.cmp <- Pervasives.compare is executed, t1.cmp != t2.cmp because two different closures have been created. Instead, first define let compare = fun x y -> Pervasives.compare x y in ....

```
type ('a, 'b) t = {
  compare : 'a -> 'a -> int ;
  map : ('a, 'b) Mappe.t ;
}
```

```
val is_empty : ('a, 'b) t -> bool
```

```
val empty : ('a -> 'a -> int) -> ('a, 'b) t
```

```
val add : 'a -> 'b -> ('a, 'b) t -> ('a, 'b) t
```

```
val find : 'a -> ('a, 'b) t -> 'b
```

```
val remove : 'a -> ('a, 'b) t -> ('a, 'b) t
```

```
val mem : 'a -> ('a, 'b) t -> bool
```

```
val addmap : ('a, 'b) t -> ('a, 'b) t -> ('a, 'b) t
```

```
val merge : ('a -> 'a -> 'a) ->
  ('b, 'a) t -> ('b, 'a) t -> ('b, 'a) t
```

```
val mergei :
  ('a -> 'b -> 'b -> 'b) ->
  ('a, 'b) t -> ('a, 'b) t -> ('a, 'b) t
```

```
val common : ('a -> 'b -> 'c) ->
  ('d, 'a) t -> ('d, 'b) t -> ('d, 'c) t
```

```
val commoni :
  ('a -> 'b -> 'c -> 'd) ->
  ('a, 'b) t -> ('a, 'c) t -> ('a, 'd) t
```

```
val combine :
  ('a -> 'b option -> 'c option -> 'd option) ->
  ('a, 'b) t -> ('a, 'c) t -> ('a, 'd) t
```

```
val interset : ('a, 'b) t -> 'a PSette.t -> ('a, 'b) t
val diffset : ('a, 'b) t -> 'a PSette.t -> ('a, 'b) t
val iter : ('a -> 'b -> unit) -> ('a, 'b) t -> unit
val map : ('a -> 'b) -> ('c, 'a) t -> ('c, 'b) t
val mapi : ('a -> 'b -> 'c) -> ('a, 'b) t -> ('a, 'c) t
val fold : ('a -> 'b -> 'c -> 'c) -> ('a, 'b) t -> 'c -> 'c
val maptoset : ('a, 'b) t -> 'a PSette.t
val mapofset : ('a -> 'b) -> 'a PSette.t -> ('a, 'b) t
val compare : ('a -> 'b -> int) -> ('c, 'a) t -> ('c, 'b) t -> int
val comparei : ('a -> 'b -> 'c -> int) -> ('a, 'b) t -> ('a, 'c) t -> int
val equal : ('a -> 'b -> bool) -> ('c, 'a) t -> ('c, 'b) t -> bool
val equali : ('a -> 'b -> 'c -> bool) -> ('a, 'b) t -> ('a, 'c) t -> bool
val subset : ('a -> 'b -> bool) -> ('c, 'a) t -> ('c, 'b) t -> bool
val subseti : ('a -> 'b -> 'c -> bool) -> ('a, 'b) t -> ('a, 'c) t -> bool
val filter : ('a -> 'b -> bool) -> ('a, 'b) t -> ('a, 'b) t
val partition : ('a -> 'b -> bool) ->
  ('a, 'b) t -> ('a, 'b) t * ('a, 'b) t
val cardinal : ('a, 'b) t -> int
val bindings : ('a, 'b) t -> ('a * 'b) list
val min_key : ('a, 'b) t -> 'a
val max_key : ('a, 'b) t -> 'a
val choose : ('a, 'b) t -> 'a * 'b
val print :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  ?firstbind:(unit, Format.formatter, unit) Pervasives.format ->
  ?sepbind:(unit, Format.formatter, unit) Pervasives.format ->
  ?lastbind:(unit, Format.formatter, unit) Pervasives.format ->
  (Format.formatter -> 'a -> unit) ->
  (Format.formatter -> 'b -> unit) ->
  Format.formatter -> ('a, 'b) t -> unit
```

Chapter 6

Module Hashhe : Hash tables and hash functions (extension of standard library module)

Hash tables are hashed association tables, with in-place modification.

Modified by B. Jeannet: functions `map`, `copy` and `print`.

```
type ('a, 'b) hashtbl
```

```
type 'a compare = {
  hash : 'a -> int ;
  equal : 'a -> 'a -> bool ;
}
```

Generic interface

```
type ('a, 'b) t = ('a, 'b) hashtbl
```

The type of hash tables from type `'a` to type `'b`.

```
val create : int -> ('a, 'b) t
```

`create n` creates a new, empty hash table, with initial size `n`. For best results, `n` should be on the order of the expected number of elements that will be in the table. The table grows as needed, so `n` is just an initial guess.

```
val clear : ('a, 'b) t -> unit
```

Empty a hash table.

```
val add : ('a, 'b) t -> 'a -> 'b -> unit
```

`add tbl x y` adds a binding of `x` to `y` in table `tbl`. Previous bindings for `x` are not removed, but simply hidden. That is, after performing `Hashhe.remove`[6] `tbl x`, the previous binding for `x`, if any, is restored. (Same behavior as with association lists.)

```
val copy : ('a, 'b) t -> ('a, 'b) t
```

Return a copy of the given hashtable.

```
val find : ('a, 'b) t -> 'a -> 'b
```

`find tbl x` returns the current binding of `x` in `tbl`, or raises `Not_found` if no such binding exists.

```
val find_all : ('a, 'b) t -> 'a -> 'b list
```

`find_all tbl x` returns the list of all data associated with `x` in `tbl`. The current binding is returned first, then the previous bindings, in reverse order of introduction in the table.

```
val mem : ('a, 'b) t -> 'a -> bool
```

`mem tbl x` checks if `x` is bound in `tbl`.

```
val remove : ('a, 'b) t -> 'a -> unit
```

`remove tbl x` removes the current binding of `x` in `tbl`, restoring the previous binding if it exists. It does nothing if `x` is not bound in `tbl`.

```
val replace : ('a, 'b) t -> 'a -> 'b -> unit
```

`replace tbl x y` replaces the current binding of `x` in `tbl` by a binding of `x` to `y`. If `x` is unbound in `tbl`, a binding of `x` to `y` is added to `tbl`. This is functionally equivalent to `Hashhe.remove`[6] `tbl x` followed by `Hashhe.add`[6] `tbl x y`.

```
val iter : ('a -> 'b -> unit) -> ('a, 'b) t -> unit
```

`iter f tbl` applies `f` to all bindings in table `tbl`. `f` receives the key as first argument, and the associated value as second argument. Each binding is presented exactly once to `f`. The order in which the bindings are passed to `f` is unspecified. However, if the table contains several bindings for the same key, they are passed to `f` in reverse order of introduction, that is, the most recent binding is passed first.

```
val fold : ('a -> 'b -> 'c -> 'c) -> ('a, 'b) t -> 'c -> 'c
```

`fold f tbl init` computes `(f kN dN ...  (f k1 d1 init)...)`, where `k1 ...  kN` are the keys of all bindings in `tbl`, and `d1 ...  dN` are the associated values. Each binding is presented exactly once to `f`. The order in which the bindings are passed to `f` is unspecified. However, if the table contains several bindings for the same key, they are passed to `f` in reverse order of introduction, that is, the most recent binding is passed first.

```
val map : ('a -> 'b -> 'c) -> ('a, 'b) t -> ('a, 'c) t
```

`map f tbl` applies `f` to all bindings in table `tbl` and creates a new hashtable associating the results of `f` to the same key type. `f` receives the key as first argument, and the associated value as second argument. Each binding is presented exactly once to `f`. The order in which the bindings are passed to `f` is unspecified. However, if the table contains several bindings for the same key, they are passed to `f` in reverse order of introduction, that is, the most recent binding is passed first.

```
val length : ('a, 'b) t -> int
```

`length tbl` returns the number of bindings in `tbl`. Multiple bindings are counted multiply, so `length` gives the number of times `iter` calls it first argument.

```
val print :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  ?firstbind:(unit, Format.formatter, unit) Pervasives.format ->
  ?sepbind:(unit, Format.formatter, unit) Pervasives.format ->
  ?lastbind:(unit, Format.formatter, unit) Pervasives.format ->
  (Format.formatter -> 'a -> unit) ->
  (Format.formatter -> 'b -> unit) ->
  Format.formatter -> ('a, 'b) t -> unit
```

Functorial interface

```
module type HashedType =
  sig
```

```
    type t
```

The type of the hashtable keys.

```
    val equal : t -> t -> bool
```

The equality predicate used to compare keys.

```
    val hash : t -> int
```

A hashing function on keys. It must be such that if two keys are equal according to `equal`, then they have identical hash values as computed by `hash`. Examples: suitable (`equal`, `hash`) pairs for arbitrary key types include (`(=)`, `Hashhe.HashedType.hash`[6]) for comparing objects by structure, (`(fun x y -> compare x y = 0)`, `Hashhe.HashedType.hash`[6]) for comparing objects by structure and handling `Pervasives.nan` correctly, and (`(==)`, `Hashhe.HashedType.hash`[6]) for comparing objects by addresses (e.g. for or cyclic keys).

```
  end
```

The input signature of the functor `Hashhe.Make`[6].

```
module type S =
  sig

    type key
    type 'a t = (key, 'a) Hashhe.hashtbl
    module Hash :
    Hashhe.HashedType  with type t=key
    val create : int -> 'a t
    val clear : 'a t -> unit
    val copy : 'a t -> 'a t
    val add : 'a t -> key -> 'a -> unit
    val remove : 'a t -> key -> unit
    val find : 'a t -> key -> 'a
    val find_all : 'a t -> key -> 'a list
    val replace : 'a t -> key -> 'a -> unit
    val mem : 'a t -> key -> bool
    val iter : (key -> 'a -> unit) -> 'a t -> unit
    val fold : (key -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b
    val map : (key -> 'a -> 'b) -> 'a t -> 'b t
    val length : 'a t -> int
    val print :
      ?first:(unit, Format.formatter, unit) Pervasives.format ->
      ?sep:(unit, Format.formatter, unit) Pervasives.format ->
      ?last:(unit, Format.formatter, unit) Pervasives.format ->
      ?firstbind:(unit, Format.formatter, unit) Pervasives.format ->
      ?sepbind:(unit, Format.formatter, unit) Pervasives.format ->
      ?lastbind:(unit, Format.formatter, unit) Pervasives.format ->
      (Format.formatter -> key -> unit) ->
      (Format.formatter -> 'a -> unit) -> Format.formatter -> 'a t -> unit
  end
```

The output signature of the functor `Hashhe.Make`[6].

```
module Make :
functor (H : HashedType) -> S  with type key = H.t
```

Functor building an implementation of the hashtable structure. The functor `Make` returns a structure containing a type `key` of keys and a type `'a t` of hash tables associating data of type `'a` to keys of type `key`. The operations perform similarly to those of the generic interface, but use the hashing and equality functions specified in the functor argument `H` instead of generic equality and hashing.

The polymorphic hash primitive

```
val hash : 'a -> int
```

`hash x` associates a positive integer to any value of any type. It is guaranteed that if `x = y` or `Pervasives.compare x y = 0`, then `hash x = hash y`. Moreover, `hash` always terminates, even on cyclic structures.

```
val hash_param : int -> int -> 'a -> int
```

`hash_param n m x` computes a hash value for `x`, with the same properties as for `hash`. The two extra parameters `n` and `m` give more precise control over hashing. Hashing performs a depth-first, right-to-left traversal of the structure `x`, stopping after `n` meaningful nodes were encountered, or `m` nodes, meaningful or not, were encountered. Meaningful nodes are: integers; floating-point numbers; strings; characters; booleans; and constant constructors. Larger values of `m` and `n` means that more nodes are taken into account to compute the final hash value, and therefore collisions are less likely to happen. However, hashing takes longer. The parameters `m` and `n` govern the tradeoff between accuracy and speed.

```
val stdcompare : 'a compare
module Compare :
  sig

    val resize : 'a Hashhe.compare -> ('a, 'b) Hashhe.hashtbl -> unit
    val add : 'a Hashhe.compare -> ('a, 'b) Hashhe.hashtbl -> 'a -> 'b -> unit
    val remove : 'a Hashhe.compare -> ('a, 'b) Hashhe.hashtbl -> 'a -> unit
    val find : 'a Hashhe.compare -> ('a, 'b) Hashhe.hashtbl -> 'a -> 'b
    val find_all : 'a Hashhe.compare -> ('a, 'b) Hashhe.hashtbl -> 'a -> 'b list
    val replace :
      'a Hashhe.compare -> ('a, 'b) Hashhe.hashtbl -> 'a -> 'b -> unit
    val mem : 'a Hashhe.compare -> ('a, 'b) Hashhe.hashtbl -> 'a -> bool
  end
```

Chapter 7

Module `PHashhe` : Hash tables, parametrized polymorphic version

Same interface as `Hashhe`[6], but each hash table stores its comparison functions.

```
type 'a compare = 'a Hashhe.compare = {
  hash : 'a -> int ;
  equal : 'a -> 'a -> bool ;
}
```

```
type ('a, 'b) t = {
  compare : 'a Hashhe.compare ;
  mutable hashtbl : ('a, 'b) Hashhe.t ;
}
```

```
val stdcompare : 'a compare
```

```
val create : ('a -> int) -> ('a -> 'a -> bool) -> int -> ('a, 'b) t
```

```
val create_compare : 'a Hashhe.compare -> int -> ('a, 'b) t
```

```
val clear : ('a, 'b) t -> unit
```

```
val add : ('a, 'b) t -> 'a -> 'b -> unit
```

```
val remove : ('a, 'b) t -> 'a -> unit
```

```
val find : ('a, 'b) t -> 'a -> 'b
```

```
val find_all : ('a, 'b) t -> 'a -> 'b list
```

```
val replace : ('a, 'b) t -> 'a -> 'b -> unit
```

```
val mem : ('a, 'b) t -> 'a -> bool
```

```
val copy : ('a, 'b) t -> ('a, 'b) t
```

```
val iter : ('a -> 'b -> unit) -> ('a, 'b) t -> unit
```

```
val fold : ('a -> 'b -> 'c -> 'c) -> ('a, 'b) t -> 'c -> 'c
```

```
val map : ('a -> 'b -> 'c) -> ('a, 'b) t -> ('a, 'c) t
```

```
val length : ('a, 'b) t -> int
```

```
val print :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  ?firstbind:(unit, Format.formatter, unit) Pervasives.format ->
  ?sepbind:(unit, Format.formatter, unit) Pervasives.format ->
  ?lastbind:(unit, Format.formatter, unit) Pervasives.format ->
  (Format.formatter -> 'a -> unit) ->
```

```
(Format.formatter -> 'b -> unit) ->
Format.formatter -> ('a, 'b) t -> unit
```

Chapter 8

Module `SetteI` : Sette specialized for keys of type int

```
include Sette.S
```

Chapter 9

Module `SetteS` : Sette specialized for keys of type string

```
include Sette.S
```

Chapter 10

Module `MappeI` : Mappe specialized for keys of type int

```
include Mappe.S
```

Chapter 11

Module `MappeS` : Mappe specialized for keys of type string

```
include Mappe.S
```

Chapter 12

Module `HashheI` : Hashhe specialized for keys of type int

```
include Hashhe.S
```

Chapter 13

Module `HashheIB` : Hashhe specialized for keys of type (int,bool)

```
include Hashhe.S
```

Chapter 14

Module `HashheS` : Hashhe specialized for keys of type string

```
include Hashhe.S
```

Chapter 15

Module `DMappe` : Two-way map between two ordered data types

Functional semantics of operations. The generic interface assumess that types `'a` and `'b` should be comparable with the standard `Pervasives.compare` function.

```
type ('a, 'b) t
```

The type of two-way maps

```
val mapx : ('a, 'b) t -> ('a, 'b) Mappe.t
val mapy : ('a, 'b) t -> ('b, 'a) Mappe.t
```

Return the correspondance map resp. from x to y and from y to x.

```
val empty : ('a, 'b) t
```

Empty map

```
val add : 'a -> 'b -> ('a, 'b) t -> ('a, 'b) t
```

Add a new binding to the current map and return the new map.

```
val y_of_x : 'a -> ('a, 'b) t -> 'b
```

Association

```
val x_of_y : 'a -> ('b, 'a) t -> 'b
```

Inverse association

```
val remove : 'a -> ('a, 'b) t -> ('a, 'b) t
```

Remove a binding defined by its first element and return the new map.

```
val memx : 'a -> ('a, 'b) t -> bool
```

Is the object in the map ?

```
val memy : 'a -> ('b, 'a) t -> bool
```

Is the object in the map ?

```
val merge : ('a, 'b) t -> ('a, 'b) t -> ('a, 'b) t
```

Merge the two double associations. If a key is bound to different data in the two arguments, raise `Failure`.

```
val common : ('a, 'b) t -> ('a, 'b) t -> ('a, 'b) t
```

Return the common bindings. If a key is bound to different data in the two arguments, raise `Failure`.

```
val intersetx : ('a, 'b) t -> 'a Sette.t -> ('a, 'b) t
```

Select the two-way bindings `x <->y` with `x` in the set

```
val intersety : ('a, 'b) t -> 'b Sette.t -> ('a, 'b) t
```

Select the two-way bindings `x <->y` with `y` in the set

```
val diffsetx : ('a, 'b) t -> 'a Sette.t -> ('a, 'b) t
```

Remove the two-way bindings `x <->y` with `x` in the set

```
val diffsety : ('a, 'b) t -> 'b Sette.t -> ('a, 'b) t
```

Remove the two-way bindings `x <->y` with `y` in the set

```
val iter : ('a -> 'b -> unit) -> ('a, 'b) t -> unit
```

Iterate on bindings.

```
val fold : ('a -> 'b -> 'c -> 'c) -> ('a, 'b) t -> 'c -> 'c
```

Iterate on bindings and accumulating a result.

```
val setx : ('a, 'b) t -> 'a Sette.t
```

Return the set of all objects in the first place of bindings.

```
val sety : ('a, 'b) t -> 'b Sette.t
```

Return the set of all objects in the second place of bindings.

```
val equalx : ('a, 'b) t -> ('a, 'b) t -> bool
val equaly : ('a, 'b) t -> ('a, 'b) t -> bool
val subsetx : ('a, 'b) t -> ('a, 'b) t -> bool
val subsety : ('a, 'b) t -> ('a, 'b) t -> bool
```

Test two two-way association for equality and inclusion. the `x` and `y` variants resp. exploit the tables `x->y` and `y->x`. For (slight) efficieny reason, one can choose one or the other variant.

```
val cardinal : ('a, 'b) t -> int
```

Return the number of bindings.

```
val print :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  ?firstbind:(unit, Format.formatter, unit) Pervasives.format ->
  ?sepbind:(unit, Format.formatter, unit) Pervasives.format ->
  ?lastbind:(unit, Format.formatter, unit) Pervasives.format ->
  (Format.formatter -> 'a -> unit) ->
  (Format.formatter -> 'b -> unit) ->
  Format.formatter -> ('a, 'b) t -> unit
```

Print the set of bindings.

```
module type Param =
  sig
```

```
    module MappeX :
    Mappe.S
    module MappeY :
    Mappe.S
  end
```

Input signature of the functor `DMappe.Make`[15].

```
module type S =
  sig
    module MappeX :
    Mappe.S
    module MappeY :
    Mappe.S
    type x = MappeX.key
    type y = MappeY.key
    type t
    val mapx : t -> y MappeX.t
    val mapy : t -> x MappeY.t
    val is_empty : t -> bool
    val empty : t
    val add : x -> y -> t -> t
    val y_of_x : x -> t -> y
    val x_of_y : y -> t -> x
    val remove : x -> t -> t
    val memx : x -> t -> bool
    val memy : y -> t -> bool
    val merge : t -> t -> t
    val common : t -> t -> t
    val intersetx : t -> MappeX.Setkey.t -> t
    val intersety : t -> MappeY.Setkey.t -> t
    val diffsetx : t -> MappeX.Setkey.t -> t
    val diffsety : t -> MappeY.Setkey.t -> t
    val iter : (x -> y -> unit) -> t -> unit
    val fold : (x -> y -> 'a -> 'a) -> t -> 'a -> 'a
    val setx : t -> MappeX.Setkey.t
    val sety : t -> MappeY.Setkey.t
    val equalx : t -> t -> bool
    val equaly : t -> t -> bool
    val subsetx : t -> t -> bool
    val subsety : t -> t -> bool
    val cardinal : t -> int
    val print :
      ?first:(unit, Format.formatter, unit) Pervasives.format ->
      ?sep:(unit, Format.formatter, unit) Pervasives.format ->
      ?last:(unit, Format.formatter, unit) Pervasives.format ->
      ?firstbind:(unit, Format.formatter, unit) Pervasives.format ->
      ?sepbind:(unit, Format.formatter, unit) Pervasives.format ->
      ?lastbind:(unit, Format.formatter, unit) Pervasives.format ->
      (Format.formatter -> x -> unit) ->
      (Format.formatter -> y -> unit) ->
      Format.formatter -> t -> unit
  end
```

Output signature of the functor `DMappe.Make`[15].

```
module Make :
functor (P : Param) -> S  with module MappeX = P.MappeX and module MappeY = P.MappeY
```

Functor building an implementation of the DMappe structure given two map structures.

Chapter 16

Module PDMappe : Two-way map between two ordered data types, parametrized polymorphic version

Same interface as DMappe[15].

```
type ('a, 'b) t
val mapx : ('a, 'b) t -> ('a, 'b) PMappe.t
val mapy : ('a, 'b) t -> ('b, 'a) PMappe.t
val is_empty : ('a, 'b) t -> bool
val empty : ('a -> 'a -> int) -> ('b -> 'b -> int) -> ('a, 'b) t
val add : 'a -> 'b -> ('a, 'b) t -> ('a, 'b) t
val y_of_x : 'a -> ('a, 'b) t -> 'b
val x_of_y : 'a -> ('b, 'a) t -> 'b
val remove : 'a -> ('a, 'b) t -> ('a, 'b) t
val removex : 'a -> ('a, 'b) t -> ('a, 'b) t
val removey : 'a -> ('b, 'a) t -> ('b, 'a) t
val memx : 'a -> ('a, 'b) t -> bool
val memy : 'a -> ('b, 'a) t -> bool
val merge : ('a, 'b) t -> ('a, 'b) t -> ('a, 'b) t
val common : ('a, 'b) t -> ('a, 'b) t -> ('a, 'b) t
val intersetx : ('a, 'b) t -> 'a PSette.t -> ('a, 'b) t
val intersety : ('a, 'b) t -> 'b PSette.t -> ('a, 'b) t
val diffsetx : ('a, 'b) t -> 'a PSette.t -> ('a, 'b) t
val diffsety : ('a, 'b) t -> 'b PSette.t -> ('a, 'b) t
val iter : ('a -> 'b -> unit) -> ('a, 'b) t -> unit
val fold : ('a -> 'b -> 'c -> 'c) -> ('a, 'b) t -> 'c -> 'c
val setx : ('a, 'b) t -> 'a PSette.t
val sety : ('a, 'b) t -> 'b PSette.t
val equalx : ('a, 'b) t -> ('a, 'b) t -> bool
val equaly : ('a, 'b) t -> ('a, 'b) t -> bool
val subsetx : ('a, 'b) t -> ('a, 'b) t -> bool
val subsety : ('a, 'b) t -> ('a, 'b) t -> bool
val cardinal : ('a, 'b) t -> int
```

```
val print :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  ?firstbind:(unit, Format.formatter, unit) Pervasives.format ->
  ?sepbind:(unit, Format.formatter, unit) Pervasives.format ->
  ?lastbind:(unit, Format.formatter, unit) Pervasives.format ->
  (Format.formatter -> 'a -> unit) ->
  (Format.formatter -> 'b -> unit) ->
  Format.formatter -> ('a, 'b) t -> unit
```

Chapter 17

Module DHashhe : Two-way hashtable between two data types

```
type ('a, 'b) t = {
  xy : ('a, 'b) Hashhe.t ;
  yx : ('b, 'a) Hashhe.t ;
}
```

The type of two-way hashtables, meant to be abstract

```
val hashx : ('a, 'b) t -> ('a, 'b) Hashhe.t
val hashy : ('a, 'b) t -> ('b, 'a) Hashhe.t
```

Return the correspondance hashtable resp. from x to y and from y to x. Never modify it !!!

```
val clear : ('a, 'b) t -> unit
```

Clear a table

```
val create : int -> ('a, 'b) t
```

Create a new table, with the specified initial size

```
val add : ('a, 'b) t -> 'a -> 'b -> unit
```

Add a new binding to the table.

```
val y_of_x : ('a, 'b) t -> 'a -> 'b
```

Association.

```
val x_of_y : ('a, 'b) t -> 'b -> 'a
```

Inverse association.

```
val removex : ('a, 'b) t -> 'a -> unit
```

Remove a binding defined by its first element.

```
val removey : ('a, 'b) t -> 'b -> unit
```

Remove a binding defined by its second element.

```
val memx : ('a, 'b) t -> 'a -> bool
```

Is the object registered ?

```
val memy : ('a, 'b) t -> 'b -> bool
```

Is the object registered ?

```
val iter : ('a, 'b) t -> ('a -> 'b -> unit) -> unit
```

Iterate on bindings.

```
val fold : ('a, 'b) t -> 'c -> ('a -> 'b -> 'c -> 'c) -> 'c
```

Iterate on bindings and accumulating a result.

```
val cardinal : ('a, 'b) t -> int
```

Return the number of bindings.

```
val print :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  ?firstbind:(unit, Format.formatter, unit) Pervasives.format ->
  ?sepbind:(unit, Format.formatter, unit) Pervasives.format ->
  ?lastbind:(unit, Format.formatter, unit) Pervasives.format ->
  (Format.formatter -> 'a -> unit) ->
  (Format.formatter -> 'b -> unit) ->
  Format.formatter -> ('a, 'b) t -> unit
```

Print the set of bindings.

```
module type Param =
  sig

     module HashX :
     Hashhe.S
```

Hashtable for objects in the first place of bindings

```
     module HashY :
     Hashhe.S
```

Hashtable for objects in the second place of bindings

```
  end
```

Input signature of the functor `DHashhe.Make`[17].

```
module type S =
  sig

     module HashX :
     Hashhe.S
     module HashY :
     Hashhe.S
     type x = HashX.key
     type y = HashY.key
     type t
     val hashx : t -> y HashX.t
     val hashy : t -> x HashY.t
     val clear : t -> unit
     val create : int -> t
```

```
    val add : t -> x -> y -> unit
    val y_of_x : t -> x -> y
    val x_of_y : t -> y -> x
    val removex : t -> x -> unit
    val removey : t -> y -> unit
    val memx : t -> x -> bool
    val memy : t -> y -> bool
    val iter : t -> (x -> y -> unit) -> unit
    val fold : t -> 'a -> (x -> y -> 'a -> 'a) -> 'a
    val cardinal : t -> int
    val print :
      ?first:(unit, Format.formatter, unit) Pervasives.format ->
      ?sep:(unit, Format.formatter, unit) Pervasives.format ->
      ?last:(unit, Format.formatter, unit) Pervasives.format ->
      ?firstbind:(unit, Format.formatter, unit) Pervasives.format ->
      ?sepbind:(unit, Format.formatter, unit) Pervasives.format ->
      ?lastbind:(unit, Format.formatter, unit) Pervasives.format ->
      (Format.formatter -> x -> unit) ->
      (Format.formatter -> y -> unit) ->
      Format.formatter -> t -> unit
  end
```

Output signature of the functor `DHashhe.Make`[17].

```
module Make :
functor (P : Param) -> S  with module HashX = P.HashX and module HashY = P.HashY
```

Functor building an implementation of the DHashtbl structure given two hashtables

Chapter 18

Module PDHashhe : Two-way hashtable between two data types, parametrized polymorpphic version

Same interface as DHashhe[17], but each hash table stores its comparison functions.

```
type ('a, 'b) t = {
  xy : ('a, 'b) PHashhe.t ;
  yx : ('b, 'a) PHashhe.t ;
}
val hashx : ('a, 'b) t -> ('a, 'b) PHashhe.t
val hashy : ('a, 'b) t -> ('b, 'a) PHashhe.t
val clear : ('a, 'b) t -> unit
val create_compare :
  'a Hashhe.compare -> 'b Hashhe.compare -> int -> ('a, 'b) t
val create :
  ('a -> int) ->
  ('a -> 'a -> bool) ->
  ('b -> int) -> ('b -> 'b -> bool) -> int -> ('a, 'b) t
val add : ('a, 'b) t -> 'a -> 'b -> unit
val y_of_x : ('a, 'b) t -> 'a -> 'b
val x_of_y : ('a, 'b) t -> 'b -> 'a
val removex : ('a, 'b) t -> 'a -> unit
val removey : ('a, 'b) t -> 'b -> unit
val memx : ('a, 'b) t -> 'a -> bool
val memy : ('a, 'b) t -> 'b -> bool
val iter : ('a, 'b) t -> ('a -> 'b -> unit) -> unit
val fold : ('a, 'b) t -> 'c -> ('a -> 'b -> 'c -> 'c) -> 'c
val cardinal : ('a, 'b) t -> int
val print :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  ?firstbind:(unit, Format.formatter, unit) Pervasives.format ->
  ?sepbind:(unit, Format.formatter, unit) Pervasives.format ->
```

```
?lastbind:(unit, Format.formatter, unit) Pervasives.format ->
(Format.formatter -> 'a -> unit) ->
(Format.formatter -> 'b -> unit) ->
Format.formatter -> ('a, 'b) t -> unit
```

Chapter 19

Module `SetList` : Sets over totally ordered type with lists.

All operations over sets are purely applicative (no side-effects).

```
type 'a t
```

The type of sets over elements of type `'a`.

```
val print :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  (Format.formatter -> 'a -> unit) -> Format.formatter -> 'a t -> unit
```

Printing function

```
val empty : 'a t
```

The empty set.

```
val is_empty : 'a t -> bool
```

Test whether a set is empty or not.

```
val mem : 'a -> 'a t -> bool
```

`mem x s` tests whether `x` belongs to the set `s`.

```
val of_list : 'a list -> 'a t
```

Conversion from a list (unsafe operation)

```
val to_list : 'a t -> 'a list
```

Conversion to a list

```
val singleton : 'a -> 'a t
```

`singleton x` returns the one-element set containing only `x`.

```
val add : 'a -> 'a t -> 'a t
```

`add x s` returns a set containing all elements of `s`, plus `x`. If `x` was already in `s`, `s` is returned unchanged.

```
val remove : 'a -> 'a t -> 'a t
```

`remove x s` returns a set containing all elements of `s`, except `x`. If `x` was not in `s`, `s` is returned unchanged.

`val union : 'a t -> 'a t -> 'a t`

`val inter : 'a t -> 'a t -> 'a t`

`val diff : 'a t -> 'a t -> 'a t`

Union, intersection and set difference.

`val compare : 'a t -> 'a t -> int`

Total ordering between sets. Can be used as the ordering function for doing sets of sets.

`val equal : 'a t -> 'a t -> bool`

`equal s1 s2` tests whether the sets `s1` and `s2` are equal, that is, contain equal elements.

`val subset : 'a t -> 'a t -> bool`

`subset s1 s2` tests whether the set `s1` is a subset of the set `s2`.

`val iter : ('a -> unit) -> 'a t -> unit`

`iter f s` applies `f` in turn to all elements of `s`. The order in which the elements of `s` are presented to `f` is in ascending order.

`val fold : ('a -> 'b -> 'b) -> 'a t -> 'b -> 'b`

`fold f s a` computes `(f xN ... (f x2 (f x1 a))...)`, where `x1 ... xN` are the elements of `s`. The order in which elements of `s` are presented to `f` is in ascending order.

Raises `Not_found` if no fount

Returns the computed accumulator

`val fold_right : ('a -> 'b -> 'b) -> 'a t -> 'b -> 'b`

`val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b t -> 'a`

Idem as `List.fold_X` functions

`val cardinal : 'a t -> int`

Return the number of elements of a set.

`val elements : 'a t -> 'a list`

Return the list of all elements of the given set. The returned list is sorted in increasing order with respect to the ordering `Pervasives.compare`.

`val min_elt : 'a t -> 'a`

Return the smallest element of the given set (with respect to the `Pervasives.compare` ordering), or raise `Not_found` if the set is empty.

`val max_elt : 'a t -> 'a`

Same as `min_elt`, but returns the largest element of the given set.

`val choose : 'a t -> 'a`

Return one element of the given set, or raise `Not_found` if the set is empty. Which element is chosen is unspecified, but equal elements will be chosen for equal sets.

`val filter : ('a -> bool) -> 'a t -> 'a t`

`filter p s` returns the set of all elements in `s` that satisfy predicate `p`.

`val partition : ('a -> bool) -> 'a t -> 'a t * 'a t`

`partition p s` returns a pair of sets `(s1, s2)`, where `s1` is the set of all the elements of `s` that satisfy the predicate `p`, and `s2` is the set of all the elements of `s` that do not satisfy `p`.

`val exists : ('a -> bool) -> 'a t -> bool`

`exists p s` checks if at least one element of the set satisfies the predicate `p`.

`val for_all : ('a -> bool) -> 'a t -> bool`

`for_all p s` checks if all elements of the set satisfy the predicate `p`.

```
module type S =
  sig
```

`type elt`

The type of the set elements.

`type t`

The type of sets.

```
val print :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  (Format.formatter -> elt -> unit) ->
  Format.formatter -> t -> unit
```

Printing function

`val empty : t`

The empty set.

`val is_empty : t -> bool`

Test whether a set is empty or not.

`val mem : elt -> t -> bool`

`mem x s` tests whether `x` belongs to the set `s`.

`val of_list : elt list -> t`

Conversion from a list (unsafe operation)

`val to_list : t -> elt list`

Conversion to a list

`val singleton : elt -> t`

`singleton x` returns the one-element set containing only `x`.

`val add : elt -> t -> t`

`add x s` returns a set containing all elements of `s`, plus `x`. If `x` was already in `s`, `s` is returned unchanged.

`val remove : elt -> t -> t`

`remove x s` returns a set containing all elements of `s`, except `x`. If `x` was not in `s`, `s` is returned unchanged.

`val union : t -> t -> t`

`val inter : t -> t -> t`

`val diff : t -> t -> t`

Union, intersection and set difference.

`val compare : t -> t -> int`

Total ordering between sets. Can be used as the ordering function for doing sets of sets.

`val equal : t -> t -> bool`

`equal s1 s2` tests whether the sets `s1` and `s2` are equal, that is, contain equal elements.

`val subset : t -> t -> bool`

`subset s1 s2` tests whether the set `s1` is a subset of the set `s2`.

`val iter : (elt -> unit) -> t -> unit`

`iter f s` applies `f` in turn to all elements of `s`. The order in which the elements of `s` are presented to `f` is in ascending order.

`val fold : (elt -> 'a -> 'a) -> t -> 'a -> 'a`

`fold f s a` computes `(f xN ... (f x2 (f x1 a))...)`, where `x1 ... xN` are the elements of `s`. The order in which elements of `s` are presented to `f` is in ascending order.
Raises `Not_found` if no fount
Returns the computed accumulator

`val fold_right : (elt -> 'a -> 'a) -> t -> 'a -> 'a`

`val fold_left : ('a -> elt -> 'a) -> 'a -> t -> 'a`

Idem as `List.fold_X` functions

`val cardinal : t -> int`

Return the number of elements of a set.

`val elements : t -> elt list`

Return the list of all elements of the given set. The returned list is sorted in increasing order.

`val min_elt : t -> elt`

Return the smallest element of the given set (with respect to the `Ord.compare` ordering), or raise `Not_found` if the set is empty.

`val max_elt : t -> elt`

Same as `min_elt`, but returns the largest element of the given set.

`val choose : t -> elt`

Return one element of the given set, or raise `Not_found` if the set is empty. Which element is chosen is unspecified, but equal elements will be chosen for equal sets.

`val filter : (elt -> bool) -> t -> t`

`filter p s` returns the set of all elements in `s` that satisfy predicate `p`.

```
val partition : (elt -> bool) -> t -> t * t
```

`partition p s` returns a pair of sets `(s1, s2)`, where `s1` is the set of all the elements of `s` that satisfy the predicate `p`, and `s2` is the set of all the elements of `s` that do not satisfy `p`.

```
val exists : (elt -> bool) -> t -> bool
```

`exists p s` checks if at least one element of the set satisfies the predicate `p`.

```
val for_all : (elt -> bool) -> t -> bool
```

`for_all p s` checks if all elements of the set satisfy the predicate `p`.

```
end
```

Output signature of the functor `SetList.Make`[19]

```
module Make :
functor (Ord :  Set.OrderedType) -> S  with type elt = Ord.t
```

Functor building an implementation of the SetList structure given a totally ordered type.

Chapter 20

Module `MultiSetList` : Multisets implemented with lists

`type 'a t`

The type of multisets over elements of type 'a.

`val of_set : 'a SetList.t -> 'a t`

Conversion from sets of module `SetList`.

`val to_set : 'a t -> 'a SetList.t`

`to_set m` returns the set of elements in the multiset `m`.

`val empty : 'a t`

The empty multiset.

`val is_empty : 'a t -> bool`

Test whether a multiset is empty or not.

`val mem : 'a -> 'a t -> bool`

`mem x s` tests whether `x` belongs to the multiset `s`.

`val mult : 'a -> 'a t -> int`

`mult elt mset` returns the number of occurences of the element `elt` in the multiset `mset`.

`val singleton : 'a * int -> 'a t`

`singleton x` returns the one-element multiset containing only `x`.

`val add : 'a * int -> 'a t -> 'a t`

`add x s` returns a multiset containing all elements of `s`, plus `x`. If `x` was already in `s`, its occurence number is incremented.

`val remove : 'a * int -> 'a t -> 'a t`

`remove x s` returns a multiset containing all elements of `s`, with the occurence number of `x` decremented. If it becomes `0`, `x` is not anymore in `s`. If `x` was not in `s`, `s` is returned unchanged.

`val union : 'a t -> 'a t -> 'a t`

`val inter : 'a t -> 'a t -> 'a t`

`val diff : 'a t -> 'a t -> 'a t`

Union, intersection and multiset difference.

`val union_set : 'a t -> 'a SetList.t -> 'a t`

`val inter_set : 'a t -> 'a SetList.t -> 'a t`

`val diff_set : 'a t -> 'a SetList.t -> 'a t`

Union, intersection and multiset difference with a set.

`val compare : 'a t -> 'a t -> int`

Total ordering between multisets. Can be used as the ordering function for doing sets of multisets.

`val equal : 'a t -> 'a t -> bool`

`equal s1 s2` tests whether the multisets `s1` and `s2` are equal, that is, contain equal elements with equal occurence numbers.

`val subset : 'a t -> 'a t -> bool`

`subset s1 s2` tests whether the multiset `s1` is a subset of the multiset `s2`.

`val iter : ('a * int -> unit) -> 'a t -> unit`

`iter f s` applies `f` in turn to all elements of `s`, with their occurence number. The order in which the elements of `s` are presented to `f` is in ascending order.

`val fold : ('a * int -> 'b -> 'b) -> 'a t -> 'b -> 'b`

`fold f s a` computes `(f (xN,On) ... (f (x2,O2) (f (x1,o1) a))...)`, where `(x1,o1) ... (xN,oN)` are pairs of elements of `s` with their occurence number. The order in which elements of `s` are presented to `f` is in ascending order.

`val fold_right : ('a * int -> 'b -> 'b) -> 'a t -> 'b -> 'b`

`val fold_left : ('a -> 'b * int -> 'a) -> 'a -> 'b t -> 'a`

Idem as `List.fold_X` functions

`val filter : ('a -> bool) -> 'a t -> 'a t`

`filter p l` returns all the elements of the multiset `l` that satisfy the predicate `p`.

`val partition : ('a -> bool) -> 'a t -> 'a t * 'a t`

`partition p l` returns a pair of multisets `(l1, l2)`, where `l1` is the multiset of all the elements of `l` that satisfy the predicate `p`, and `l2` is the multiset of all the elements of `l` that do not satisfy `p`.

`val cardinal : 'a t -> int`

Return the number of elements of a multiset.

`val elements : 'a t -> 'a SetList.t`

Return the list of all elements of the given multiset. The returned list is sorted in increasing order with respect to the ordering `Pervasives.compare`.

`val min_elt : 'a t -> 'a`

Return the smallest element of the given multiset (with respect to the `Pervasives.compare` ordering), or raise `Not_found` if the multiset is empty.

`val max_elt : 'a t -> 'a`

Same as `min_elt`, but returns the largest element of the given multiset.

```
val min : 'a t -> 'a * int
```

Return an element with the minimum occurence number, or raise `Not_found` if the multiset is empty. Which element is chosen is unspecified, but equal elements will be chosen for equal multisets.

```
val max : 'a t -> 'a * int
```

Return an element with the maximum occurence number, or raise `Not_found` if the multiset is empty. Which element is chosen is unspecified, but equal elements will be chosen for equal multisets.

```
val mins : 'a t -> 'a SetList.t * int
```

Return the set of elements with the minimum occurence number, or raise `Not_found` if the multiset is empty.

```
val maxs : 'a t -> 'a SetList.t * int
```

Return the set of elements with the maximum occurence number, or raise `Not_found` if the multiset is empty.

```
val choose : 'a t -> 'a
```

Return one element of the given multiset, or raise `Not_found` if the multiset is empty. Which element is chosen is unspecified, but equal elements will be chosen for equal multisets.

```
val print :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  (Format.formatter -> 'a -> unit) ->
  Format.formatter -> 'a t -> unit
```

Printing function

```
module type S =
  sig
```

```
type elt
```

Type of multiset elements

```
type t
```

Type of multisets over type `elt`.

```
val empty : t
```

The empty multiset.

```
val is_empty : t -> bool
```

Test whether a multiset is empty or not.

```
val mem : elt -> t -> bool
```

`mem x s` tests whether `x` belongs to the multiset `s`.

```
val mult : elt -> t -> int
```

`mult elt mset` returns the number of occurences of the element `elt` in the multiset `mset`.

```
val singleton : elt * int -> t
```

`singleton x` returns the one-element multiset containing only `x`.

```
val add : elt * int -> t -> t
```

`add x s` returns a multiset containing all elements of `s`, plus `x`. If `x` was already in `s`, its occurence number is incremented.

```
val remove : elt * int -> t -> t
```

`remove x s` returns a multiset containing all elements of `s`, with the occurence number of `x` decremented. If it becomes `0`, `x` is not anymore in `s`. If `x` was not in `s`, `s` is returned unchanged.

```
val union : t -> t -> t
val inter : t -> t -> t
val diff : t -> t -> t
```

Union, intersection and multiset difference.

```
val union_set : t -> elt SetList.t -> t
val inter_set : t -> elt SetList.t -> t
val diff_set : t -> elt SetList.t -> t
```

Union, intersection and multiset difference with a set.

```
val compare : t -> t -> int
```

Total ordering between multisets. Can be used as the ordering function for doing sets of multisets.

```
val equal : t -> t -> bool
```

`equal s1 s2` tests whether the multisets `s1` and `s2` are equal, that is, contain equal elements with equal occurence numbers.

```
val subset : t -> t -> bool
```

`subset s1 s2` tests whether the multiset `s1` is a subset of the multiset `s2`.

```
val iter : (elt * int -> unit) -> t -> unit
```

`iter f s` applies `f` in turn to all elements of `s`, with their occurence number. The order in which the elements of `s` are presented to `f` is in ascending order.

```
val fold : (elt * int -> 'a -> 'a) -> t -> 'a -> 'a
```

`fold f s a` computes `(f (xN,On) ... (f (x2,O2) (f (x1,o1) a))...)`, where `(x1,o1) ... (xN,oN)` are pairs of elements of `s` with their occurence number. The order in which elements of `s` are presented to `f` is in ascending order.

```
val fold_right : (elt * int -> 'a -> 'a) -> t -> 'a -> 'a
val fold_left : ('a -> elt * int -> 'a) -> 'a -> t -> 'a
```

Idem as `List.fold_X` functions

```
val filter : (elt -> bool) -> t -> t
```

`filter p l` returns all the elements of the multiset `l` that satisfy the predicate `p`.

```
val partition : (elt -> bool) ->
  t -> t * t
```

`partition p l` returns a pair of multisets `(l1, l2)`, where `l1` is the multiset of all the elements of `l` that satisfy the predicate `p`, and `l2` is the multiset of all the elements of `l` that do not satisfy `p`.

```
val cardinal : t -> int
```

Return the number of elements of a multiset.

```
val elements : t -> elt SetList.t
```

Return the list of all elements of the given multiset. The returned list is sorted in increasing order with respect to the ordering `Pervasives.compare`.

```
val min_elt : t -> elt
```

Return the smallest element of the given multiset (with respect to the `Pervasives.compare` ordering), or raise `Not_found` if the multiset is empty.

```
val max_elt : t -> elt
```

Same as `min_elt`, but returns the largest element of the given multiset.

```
val min : t -> elt * int
```

Return an element with the minimum occurence number, or raise `Not_found` if the multiset is empty. Which element is chosen is unspecified, but equal elements will be chosen for equal multisets.

```
val max : t -> elt * int
```

Return an element with the maximum occurence number, or raise `Not_found` if the multiset is empty. Which element is chosen is unspecified, but equal elements will be chosen for equal multisets.

```
val mins : t -> elt SetList.t * int
```

Return the set of elements with the minimum occurence number, or raise `Not_found` if the multiset is empty.

```
val maxs : t -> elt SetList.t * int
```

Return the set of elements with the maximum occurence number, or raise `Not_found` if the multiset is empty.

```
val choose : t -> elt
```

Return one element of the given multiset, or raise `Not_found` if the multiset is empty. Which element is chosen is unspecified, but equal elements will be chosen for equal multisets.

```
val of_set : elt SetList.t -> t
```

Conversion from sets of module `SetList`.

```
val to_set : t -> elt SetList.t
```

`to_set m` returns the set of elements in the multiset `m`.

```
val print :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  (Format.formatter -> elt -> unit) ->
  Format.formatter -> t -> unit
```

Printing function

```
  end
```

Output signature of the functor `MultiSetList.Make`[20]

```
module Make :
functor (Ord :  Set.OrderedType) -> S  with type elt = Ord.t
```

Functor building an implementation of the MultiSetList structure given a totally ordered type.

Chapter 21

Module `Ilist` : Imbricated lists

The operations of this module have a functional semantics.

```
type 'a el =
  | Atome of 'a
```

Terminal case

```
  | List of 'a t
```

The element is recursively a list.

Type of list elements.

```
type 'a t = 'a el list
```

Type of imbricated lists.

```
val cons : 'a el -> 'a t -> 'a t
```

Adding a new list element at the begining of the list

```
val atome : 'a -> 'a el
```

Create a list element from a single element.

```
val list : 'a t -> 'a el
```

Create a list element from a list.

```
val of_list : 'a list -> 'a t
```

Create a recursive list from a list

```
val hd : 'a t -> 'a el
```

Return the head of the list.

```
val tl : 'a t -> 'a t
```

Return the tail of the list.

```
val length : 'a t -> int
```

Return the ength of the list.

```
val depth : 'a t -> int
```

Return the (maximal) depth of the list.

- `depth [] = 0`

- `depth [a;b;c] = 1`
- `depth [[a];b] = 2`

`val append : 'a t -> 'a t -> 'a t`

Append two lists

`val flatten : ?depth:int -> 'a t -> 'a t`

Flatten the recursive list, but only starting from the given depth. Defaut depth is 1.

- `flatten [] = []`
- `flatten [a;[b;[c];d];e;[f]] = [a;b;c;d;e;f]`
- `flatten ~depth:2 [a;[b;[c];d];e;[f]] = [a;[b;c;d];e;[f]]`
- `flatten ~depth:3 [a;[b;[c];d];e;[f]] = [a;[b;[c];d];e;[f]]`

`val rev : 'a t -> 'a t`

Recursively reverse the recursive list

- `rev [a;[b;[c];d];e;[f]] = [[f];e;[d;[c];b];a]`

`val mem : 'a -> 'a t -> bool`

Membership test.

`val exists : ('a -> bool) -> 'a t -> bool`

Existence test

`val map : (bool -> 'a -> 'b) -> 'a t -> 'b t`

Ordinary map function. The boolean value indicates whether the element is beginning a recursive list.

`val iter : (bool -> 'a -> unit) -> 'a t -> unit`

Ordinary iteration function. The boolean value indicates whether the element is beginning a recursive list.

`val fold_left : ('a -> bool -> 'b -> 'a) -> 'a -> 'b t -> 'a`

Ordinary fold function, from left to right.

`val fold_right : (bool -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b`

Ordinary fold function, from right to left.

```
val print :
  ?first:(unit, Format.formatter, unit) Pervasives.format ->
  ?sep:(unit, Format.formatter, unit) Pervasives.format ->
  ?last:(unit, Format.formatter, unit) Pervasives.format ->
  (Format.formatter -> 'a -> unit) -> Format.formatter -> 'a t -> unit
```

Printing function.

Chapter 22

Module FGraph1 : Directed graphs, functional API, with one-way information maintained,

22.1 Polymorphic version

```
type ('a, 'b, 'c, 'd) t
val info : ('a, 'b, 'c, 'd) t -> 'd
val set_info : ('a, 'b, 'c, 'd) t -> 'd -> ('a, 'b, 'c, 'd) t
val succ : ('a, 'b, 'c, 'd) t -> 'a -> 'a Sette.t
val pred : ('a, 'b, 'c, 'd) t -> 'a -> 'a Sette.t
```

Expensive operation, requires iterations on all vertices

```
val attrvertex : ('a, 'b, 'c, 'd) t -> 'a -> 'b
val attredge : ('a, 'b, 'c, 'd) t -> 'a * 'a -> 'c
val empty : 'a -> ('b, 'c, 'd, 'a) t
val size : ('a, 'b, 'c, 'd) t -> int
val is_empty : ('a, 'b, 'c, 'd) t -> bool
val is_vertex : ('a, 'b, 'c, 'd) t -> 'a -> bool
val is_edge : ('a, 'b, 'c, 'd) t -> 'a * 'a -> bool
val vertices : ('a, 'b, 'c, 'd) t -> 'a Sette.t
val edges : ('a, 'b, 'c, 'd) t -> ('a * 'a) Sette.t
val map_vertex : ('a, 'b, 'c, 'd) t -> ('a -> 'b -> 'e) -> ('a, 'e, 'c, 'd) t
val map_edge :
  ('a, 'b, 'c, 'd) t ->
  ('a * 'a -> 'c -> 'e) -> ('a, 'b, 'e, 'd) t
val map_info : ('a, 'b, 'c, 'd) t -> ('d -> 'e) -> ('a, 'b, 'c, 'e) t
val map :
  ('a, 'b, 'c, 'd) t ->
  ('a -> 'b -> 'e) ->
  ('a * 'a -> 'c -> 'f) -> ('d -> 'g) -> ('a, 'e, 'f, 'g) t
val iter_vertex :
  ('a, 'b, 'c, 'd) t -> ('a -> 'b -> 'a Sette.t -> unit) -> unit
```

```
val iter_edge : ('a, 'b, 'c, 'd) t -> ('a * 'a -> 'c -> unit) -> unit
```

```
val fold_vertex :
  ('a, 'b, 'c, 'd) t ->
  'e -> ('a -> 'b -> 'a Sette.t -> 'e -> 'e) -> 'e
```

```
val fold_edge : ('a, 'b, 'c, 'd) t -> 'e -> ('a * 'a -> 'c -> 'e -> 'e) -> 'e
```

```
val add_edge : ('a, 'b, 'c, 'd) t -> 'a * 'a -> 'c -> ('a, 'b, 'c, 'd) t
```

```
val remove_edge : ('a, 'b, 'c, 'd) t -> 'a * 'a -> ('a, 'b, 'c, 'd) t
```

```
val add_vertex : ('a, 'b, 'c, 'd) t -> 'a -> 'b -> ('a, 'b, 'c, 'd) t
```

```
val remove_vertex : ('a, 'b, 'c, 'd) t -> 'a -> ('a, 'b, 'c, 'd) t
```

Expensive operation, requires iterations on all vertices

```
val topological_sort : ('a, 'b, 'c, 'd) t -> 'a -> 'a list
```

```
val topological_sort_multi :
  'a -> ('a, 'b, 'c, 'd) t -> 'a Sette.t -> 'a list
```

```
val reachable : ('a, 'b, 'c, 'd) t -> 'a -> 'a Sette.t
```

```
val reachable_multi : 'a -> ('a, 'b, 'c, 'd) t -> 'a Sette.t -> 'a Sette.t
```

```
val cfc : ('a, 'b, 'c, 'd) t -> 'a -> 'a list list
```

```
val cfc_multi : 'a -> ('a, 'b, 'c, 'd) t -> 'a Sette.t -> 'a list list
```

```
val scfc : ('a, 'b, 'c, 'd) t -> 'a -> 'a Ilist.t
```

```
val scfc_multi : 'a -> ('a, 'b, 'c, 'd) t -> 'a Sette.t -> 'a Ilist.t
```

```
val min : ('a, 'b, 'c, 'd) t -> 'a Sette.t
```

```
val max : ('a, 'b, 'c, 'd) t -> 'a Sette.t
```

```
val print :
  (Format.formatter -> 'a -> unit) ->
  (Format.formatter -> 'b -> unit) ->
  (Format.formatter -> 'c -> unit) ->
  (Format.formatter -> 'd -> unit) ->
  Format.formatter -> ('a, 'b, 'c, 'd) t -> unit
```

22.2 Functor version

```
module type T =
  sig
```

```
    module MapV :
    Mappe.S
```

Map module for associating attributes to vertices, of type `MapV.key`

```
    module MapE :
    Mappe.S  with type key = MapV.key * MapV.key
```

Map module for associating attributes to edges, of type `MapV.key * MapV.key`

```
  end
```

```
module type S =
  sig
```

```
    type vertex
```

The type of vertices

```
module SetV :
Sette.S  with type elt=vertex
```

The type of sets of vertices

```
module SetE :
Sette.S  with type elt=vertex*vertex
```

The type of sets of edges

```
module MapV :
Mappe.S  with type key=vertex and module Setkey=SetV
```

The Map for vertices

```
module MapE :
Mappe.S  with type key=vertex*vertex and module Setkey=SetE
```

The Map for edges

```
type ('a, 'b, 'c) t
```

The type of graphs, where:

- `'b` is the type of vertex attribute (attrvertex);
- `'c` is the type of edge attributes (attredge)

```
val info : ('a, 'b, 'c) t -> 'c
val set_info : ('a, 'b, 'c) t -> 'c -> ('a, 'b, 'c) t
val succ : ('a, 'b, 'c) t -> vertex -> SetV.t
val pred : ('a, 'b, 'c) t -> vertex -> SetV.t
val attrvertex : ('a, 'b, 'c) t -> vertex -> 'a
val attredge : ('a, 'b, 'c) t -> vertex * vertex -> 'b
val empty : 'a -> ('b, 'c, 'a) t
val size : ('a, 'b, 'c) t -> int
val is_empty : ('a, 'b, 'c) t -> bool
val is_vertex : ('a, 'b, 'c) t -> vertex -> bool
val is_edge : ('a, 'b, 'c) t -> vertex * vertex -> bool
val vertices : ('a, 'b, 'c) t -> SetV.t
val edges : ('a, 'b, 'c) t -> SetE.t
val map_vertex : ('a, 'b, 'c) t ->
  (vertex -> 'a -> 'd) -> ('d, 'b, 'c) t
val map_edge :
  ('a, 'b, 'c) t ->
  (vertex * vertex -> 'b -> 'd) -> ('a, 'd, 'c) t
val map_info : ('a, 'b, 'c) t -> ('c -> 'd) -> ('a, 'b, 'd) t
val map :
  ('a, 'b, 'c) t ->
  (vertex -> 'a -> 'd) ->
  (vertex * vertex -> 'b -> 'e) ->
  ('c -> 'f) -> ('d, 'e, 'f) t
val iter_vertex :
  ('a, 'b, 'c) t ->
  (vertex -> 'a -> SetV.t -> unit) -> unit
```

```
    val iter_edge : ('a, 'b, 'c) t ->
      (vertex * vertex -> 'b -> unit) -> unit
    val fold_vertex :
      ('a, 'b, 'c) t ->
      'd -> (vertex -> 'a -> SetV.t -> 'd -> 'd) -> 'd
    val fold_edge :
      ('a, 'b, 'c) t ->
      'd -> (vertex * vertex -> 'b -> 'd -> 'd) -> 'd
    val add_edge : ('a, 'b, 'c) t ->
      vertex * vertex -> 'b -> ('a, 'b, 'c) t
    val remove_edge : ('a, 'b, 'c) t ->
      vertex * vertex -> ('a, 'b, 'c) t
    val add_vertex : ('a, 'b, 'c) t ->
      vertex -> 'a -> ('a, 'b, 'c) t
    val remove_vertex : ('a, 'b, 'c) t -> vertex -> ('a, 'b, 'c) t
    val topological_sort : ('a, 'b, 'c) t -> vertex -> vertex list
    val topological_sort_multi :
      vertex ->
      ('a, 'b, 'c) t -> SetV.t -> vertex list
    val reachable : ('a, 'b, 'c) t -> vertex -> SetV.t
    val reachable_multi : vertex ->
      ('a, 'b, 'c) t -> SetV.t -> SetV.t
    val cfc : ('a, 'b, 'c) t -> vertex -> vertex list list
    val cfc_multi : vertex ->
      ('a, 'b, 'c) t -> SetV.t -> vertex list list
    val scfc : ('a, 'b, 'c) t -> vertex -> vertex Ilist.t
    val scfc_multi : vertex ->
      ('a, 'b, 'c) t -> SetV.t -> vertex Ilist.t
    val min : ('a, 'b, 'c) t -> SetV.t
    val max : ('a, 'b, 'c) t -> SetV.t
    val print :
      (Format.formatter -> vertex -> unit) ->
      (Format.formatter -> 'a -> unit) ->
      (Format.formatter -> 'b -> unit) ->
      (Format.formatter -> 'c -> unit) ->
      Format.formatter -> ('a, 'b, 'c) t -> unit
  end
```

```
module Make :
functor (T : T) -> S  with type vertex=T.MapV.key and module SetV=T.MapV.Setkey and module SetE=T.MapE.Setkey and module MapV=T.MapV and module MapE=T.MapE
```

Chapter 23

Module FGraph : Directed graphs, functional API, with two-way information maintained,

```
type ('a, 'b) node = {
  succ : 'a Sette.t ;
  pred : 'a Sette.t ;
  attrvertex : 'b ;
}
type ('a, 'b, 'c, 'd) graph = {
  nodes : ('a, ('a, 'b) node) Mappe.t ;
  arcs : ('a * 'a, 'c) Mappe.t ;
  info : 'd ;
}
```

23.1 Polymorphic version

```
type ('a, 'b, 'c, 'd) t
val info : ('a, 'b, 'c, 'd) t -> 'd
val set_info : ('a, 'b, 'c, 'd) t -> 'd -> ('a, 'b, 'c, 'd) t
val succ : ('a, 'b, 'c, 'd) t -> 'a -> 'a Sette.t
val pred : ('a, 'b, 'c, 'd) t -> 'a -> 'a Sette.t
val attrvertex : ('a, 'b, 'c, 'd) t -> 'a -> 'b
val attredge : ('a, 'b, 'c, 'd) t -> 'a * 'a -> 'c
val empty : 'a -> ('b, 'c, 'd, 'a) t
val size_vertex : ('a, 'b, 'c, 'd) t -> int
val size_edge : ('a, 'b, 'c, 'd) t -> int
val size : ('a, 'b, 'c, 'd) t -> int * int
val is_empty : ('a, 'b, 'c, 'd) t -> bool
val is_vertex : ('a, 'b, 'c, 'd) t -> 'a -> bool
val is_edge : ('a, 'b, 'c, 'd) t -> 'a * 'a -> bool
val vertices : ('a, 'b, 'c, 'd) t -> 'a Sette.t
val edges : ('a, 'b, 'c, 'd) t -> ('a * 'a) Sette.t
val map_vertex :
  ('a, 'b, 'c, 'd) t ->
  ('a -> 'b -> pred:'a Sette.t -> succ:'a Sette.t -> 'e) ->
  ('a, 'e, 'c, 'd) t
val map_edge :
  ('a, 'b, 'c, 'd) t ->
  ('a * 'a -> 'c -> 'e) -> ('a, 'b, 'e, 'd) t
val map_info : ('a, 'b, 'c, 'd) t -> ('d -> 'e) -> ('a, 'b, 'c, 'e) t
val map :
  ('a, 'b, 'c, 'd) t ->
  ('a -> 'b -> pred:'a Sette.t -> succ:'a Sette.t -> 'e) ->
  ('a * 'a -> 'c -> 'f) -> ('d -> 'g) -> ('a, 'e, 'f, 'g) t
val transpose :
  ('a, 'b, 'c, 'd) t ->
  ('a -> 'b -> pred:'a Sette.t -> succ:'a Sette.t -> 'e) ->
  ('a * 'a -> 'c -> 'f) -> ('d -> 'g) -> ('a, 'e, 'f, 'g) t
val iter_vertex :
  ('a, 'b, 'c, 'd) t ->
  ('a -> 'b -> pred:'a Sette.t -> succ:'a Sette.t -> unit) -> unit
val iter_edge : ('a, 'b, 'c, 'd) t -> ('a * 'a -> 'c -> unit) -> unit
val fold_vertex :
  ('a, 'b, 'c, 'd) t ->
  'e -> ('a -> 'b -> pred:'a Sette.t -> succ:'a Sette.t -> 'e -> 'e) -> 'e
val fold_edge : ('a, 'b, 'c, 'd) t -> 'e -> ('a * 'a -> 'c -> 'e -> 'e) -> 'e
val add_edge : ('a, 'b, 'c, 'd) t -> 'a * 'a -> 'c -> ('a, 'b, 'c, 'd) t
val remove_edge : ('a, 'b, 'c, 'd) t -> 'a * 'a -> ('a, 'b, 'c, 'd) t
val add_vertex : ('a, 'b, 'c, 'd) t -> 'a -> 'b -> ('a, 'b, 'c, 'd) t
val remove_vertex : ('a, 'b, 'c, 'd) t -> 'a -> ('a, 'b, 'c, 'd) t
val topological_sort : ('a, 'b, 'c, 'd) t -> 'a -> 'a list
val topological_sort_multi :
  'a -> ('a, 'b, 'c, 'd) t -> 'a Sette.t -> 'a list
val reachable : ('a, 'b, 'c, 'd) t -> 'a -> 'a Sette.t
val reachable_multi : 'a -> ('a, 'b, 'c, 'd) t -> 'a Sette.t -> 'a Sette.t
val coreachable : ('a, 'b, 'c, 'd) t -> 'a -> 'a Sette.t
val coreachable_multi : 'a -> ('a, 'b, 'c, 'd) t -> 'a Sette.t -> 'a Sette.t
val cfc : ('a, 'b, 'c, 'd) t -> 'a -> 'a list list
val cfc_multi : 'a -> ('a, 'b, 'c, 'd) t -> 'a Sette.t -> 'a list list
val scfc : ('a, 'b, 'c, 'd) t -> 'a -> 'a Ilist.t
val scfc_multi : 'a -> ('a, 'b, 'c, 'd) t -> 'a Sette.t -> 'a Ilist.t
val min : ('a, 'b, 'c, 'd) t -> 'a Sette.t
val max : ('a, 'b, 'c, 'd) t -> 'a Sette.t
val print :
  (Format.formatter -> 'a -> unit) ->
  (Format.formatter -> 'b -> unit) ->
  (Format.formatter -> 'c -> unit) ->
  (Format.formatter -> 'd -> unit) ->
  Format.formatter -> ('a, 'b, 'c, 'd) t -> unit
val print_dot :
  ?titlestyle:string ->
  ?vertexstyle:string ->
  ?edgestyle:string ->
  ?title:string ->
  (Format.formatter -> 'a -> unit) ->
  (Format.formatter -> 'a -> 'b -> unit) ->
  (Format.formatter -> 'a * 'a -> 'c -> unit) ->
  Format.formatter -> ('a, 'b, 'c, 'd) t -> unit
val repr : ('a, 'b, 'c, 'd) t -> ('a, 'b, 'c, 'd) graph
val obj : ('a, 'b, 'c, 'd) graph -> ('a, 'b, 'c, 'd) t
```

23.2 Functor version

```
module type T =
  sig

    module MapV :
    Mappe.S
```

Map module for associating attributes to vertices, of type `MapV.key`

```
    module MapE :
    Mappe.S  with type key = MapV.key * MapV.key
```

Map module for associating attributes to edges, of type `MapV.key * MapV.key`

```
  end

module type S =
  sig

    type vertex
```

The type of vertices

```
    module SetV :
    Sette.S  with type elt=vertex
```

The type of sets of vertices

```
    module SetE :
    Sette.S  with type elt=vertex*vertex
```

The type of sets of edges

```
    module MapV :
    Mappe.S  with type key=vertex and module Setkey=SetV
```

The Map for vertices

```
    module MapE :
    Mappe.S  with type key=vertex*vertex and module Setkey=SetE
```

The Map for edges

```
    type ('a, 'b, 'c) t
```

The type of graphs, where:

- `'b` is the type of vertex attribute (attrvertex);

- 'c is the type of edge attributes (attredge)

```
val info : ('a, 'b, 'c) t -> 'c
```

```
val set_info : ('a, 'b, 'c) t -> 'c -> ('a, 'b, 'c) t
```

```
val succ : ('a, 'b, 'c) t -> vertex -> SetV.t
```

```
val pred : ('a, 'b, 'c) t -> vertex -> SetV.t
```

```
val attrvertex : ('a, 'b, 'c) t -> vertex -> 'a
```

```
val attredge : ('a, 'b, 'c) t -> vertex * vertex -> 'b
```

```
val empty : 'a -> ('b, 'c, 'a) t
```

```
val size_vertex : ('a, 'b, 'c) t -> int
```

```
val size_edge : ('a, 'b, 'c) t -> int
```

```
val size : ('a, 'b, 'c) t -> int * int
```

```
val is_empty : ('a, 'b, 'c) t -> bool
```

```
val is_vertex : ('a, 'b, 'c) t -> vertex -> bool
```

```
val is_edge : ('a, 'b, 'c) t -> vertex * vertex -> bool
```

```
val vertices : ('a, 'b, 'c) t -> SetV.t
```

```
val edges : ('a, 'b, 'c) t -> SetE.t
```

```
val map_vertex :
  ('a, 'b, 'c) t ->
  (vertex -> 'a -> pred:SetV.t -> succ:SetV.t -> 'd) ->
  ('d, 'b, 'c) t
```

```
val map_edge :
  ('a, 'b, 'c) t ->
  (vertex * vertex -> 'b -> 'd) -> ('a, 'd, 'c) t
```

```
val map_info : ('a, 'b, 'c) t -> ('c -> 'd) -> ('a, 'b, 'd) t
```

```
val map :
  ('a, 'b, 'c) t ->
  (vertex -> 'a -> pred:SetV.t -> succ:SetV.t -> 'd) ->
  (vertex * vertex -> 'b -> 'e) ->
  ('c -> 'f) -> ('d, 'e, 'f) t
```

```
val transpose :
  ('a, 'b, 'c) t ->
  (vertex -> 'a -> pred:SetV.t -> succ:SetV.t -> 'd) ->
  (vertex * vertex -> 'b -> 'e) ->
  ('c -> 'f) -> ('d, 'e, 'f) t
```

```
val iter_vertex :
  ('a, 'b, 'c) t ->
  (vertex ->
   'a -> pred:SetV.t -> succ:SetV.t -> unit) ->
  unit
```

```
val iter_edge : ('a, 'b, 'c) t ->
  (vertex * vertex -> 'b -> unit) -> unit
```

```
val fold_vertex :
  ('a, 'b, 'c) t ->
  'd ->
  (vertex ->
   'a -> pred:SetV.t -> succ:SetV.t -> 'd -> 'd) ->
  'd
```

```
val fold_edge :
  ('a, 'b, 'c) t ->
```

```
      'd -> (vertex * vertex -> 'b -> 'd -> 'd) -> 'd
    val add_edge : ('a, 'b, 'c) t ->
      vertex * vertex -> 'b -> ('a, 'b, 'c) t
    val remove_edge : ('a, 'b, 'c) t ->
      vertex * vertex -> ('a, 'b, 'c) t
    val add_vertex : ('a, 'b, 'c) t -> vertex -> 'a -> ('a, 'b, 'c) t
    val remove_vertex : ('a, 'b, 'c) t -> vertex -> ('a, 'b, 'c) t
    val topological_sort : ('a, 'b, 'c) t -> vertex -> vertex list
    val topological_sort_multi :
      vertex ->
      ('a, 'b, 'c) t -> SetV.t -> vertex list
    val reachable : ('a, 'b, 'c) t -> vertex -> SetV.t
    val reachable_multi : vertex ->
      ('a, 'b, 'c) t -> SetV.t -> SetV.t
    val coreachable : ('a, 'b, 'c) t -> vertex -> SetV.t
    val coreachable_multi : vertex ->
      ('a, 'b, 'c) t -> SetV.t -> SetV.t
    val cfc : ('a, 'b, 'c) t -> vertex -> vertex list list
    val cfc_multi : vertex ->
      ('a, 'b, 'c) t -> SetV.t -> vertex list list
    val scfc : ('a, 'b, 'c) t -> vertex -> vertex Ilist.t
    val scfc_multi : vertex ->
      ('a, 'b, 'c) t -> SetV.t -> vertex Ilist.t
    val min : ('a, 'b, 'c) t -> SetV.t
    val max : ('a, 'b, 'c) t -> SetV.t
    val print :
      (Format.formatter -> vertex -> unit) ->
      (Format.formatter -> 'a -> unit) ->
      (Format.formatter -> 'b -> unit) ->
      (Format.formatter -> 'c -> unit) ->
      Format.formatter -> ('a, 'b, 'c) t -> unit
    val print_dot :
      ?titlestyle:string ->
      ?vertexstyle:string ->
      ?edgestyle:string ->
      ?title:string ->
      (Format.formatter -> vertex -> unit) ->
      (Format.formatter -> vertex -> 'a -> unit) ->
      (Format.formatter -> vertex * vertex -> 'b -> unit) ->
      Format.formatter -> ('a, 'b, 'c) t -> unit
    val repr : ('a, 'b, 'c) t -> (vertex, 'a, 'b, 'c) FGraph.graph
    val obj : (vertex, 'a, 'b, 'c) FGraph.graph -> ('a, 'b, 'c) t
  end
```

```
module Make :
functor (T : T) -> S  with type vertex=T.MapV.key and module SetV=T.MapV.Setkey and module SetE=T.MapE.Setkey and module MapV=T.MapV and module MapE=T.MapE
```

Chapter 24

Module SHGraph : Oriented hypergraphs

24.1 Introduction

This module provides an abstract datatypes and functions for manipulating hypergraphs, that is, graphs where edges relates potentially more than 2 vertices. The considered hypergraphs are *oriented*: one distinguishes for a vertex incoming and outgoing hyperedges, and for an hyperedge incoming (or origin) and outgoing (or destination) vertices.

Origin and destination vertices of an hyperedge are ordered (by using arrays), in contrast with incoming and outgoing hyperedges of a vertex.

A possible use of such hypergraphs is the representation of a (fixpoint) equation system, where the unknown are the vertices and the functions the hyperedges, taking a vector of unknowns as arguments and delivering a vector of results.

A last note about the notion of connectivity, which is relevant for operations like depth-first-search, reachability and connex components notions. A destination vertex of an hyperedge is considered as reachable from an origin vertex through this hyperedge only if *all* origin vertices are reachable.

```
type 'a priority =
  | Filter of ('a -> bool)
  | Priority of ('a -> int)
```

Filtering or priority function (used by `SHGraph.topological_sort`[24.2.8.1], `SHGraph.cfc`[24.2.8.3], `SHGraph.scfc`[24.2.8.3], `SHGraph.topological_sort_multi`[24.2.8.1], `SHGraph.cfc_multi`[24.2.8.3], `SHGraph.scfc_multi`[24.2.8.3]).

`Filter p` specifies `p` as a filtering function for hyperedges: only those satisfying `p` are taken into account.

`Priority p` specifies `p` as a priority function. Hyperedges `h` with `p h < 0` are not taken into account. Otherwise, hyperedges with highest priority are explored first.

```
type ('a, 'b) compare = {
  hashv : 'a Hashhe.compare ;
  hashh : 'b Hashhe.compare ;
  comparev : 'a -> 'a -> int ;
  compareh : 'b -> 'b -> int ;
}
```

```
type ('a, 'b) vertex_n = {
  attrvertex : 'b ;
  mutable predhedge : 'a Sette.set ;
  mutable succhedge : 'a Sette.set ;
}
type ('a, 'b) hedge_n = {
  attrhedge : 'b ;
  predvertex : 'a array ;
  succvertex : 'a array ;
}
type ('a, 'b, 'c, 'd, 'e) graph = {
  vertex : ('a, ('b, 'c) vertex_n) Hashhe.hashtbl ;
  hedge : ('b, ('a, 'd) hedge_n) Hashhe.hashtbl ;
  info : 'e ;
}
```

24.2 Generic (polymorphic) interface

`val stdcompare : ('a, 'b) compare`

`type ('a, 'b, 'c, 'd, 'e) t = ('a, 'b, 'c, 'd, 'e) graph`

The type of hypergraphs where

- 'a : type of vertices
- 'b : type of hedges
- 'c : information associated to vertices
- 'd : information associated to hedges
- 'e : user-information associated to an hypergraph

`val create : int -> 'a -> ('b, 'c, 'd, 'e, 'a) t`

`create n data` creates an hypergraph, using `n` for the initial size of internal hashtables, and `data` for the user information

`val clear : ('a, 'b, 'c, 'd, 'e) t -> unit`

Remove all vertices and hyperedges of the graph.

`val is_empty : ('a, 'b, 'c, 'd, 'e) t -> bool`

Is the graph empty ?

24.2.1 Statistics

`val size_vertex : ('a, 'b, 'c, 'd, 'e) t -> int`

Number of vertices in the hypergraph

`val size_hedge : ('a, 'b, 'c, 'd, 'e) t -> int`

Number of hyperedges in the hypergraph

`val size_edgevh : ('a, 'b, 'c, 'd, 'e) t -> int`

Number of edges (vertex,hyperedge) in the hypergraph

`val size_edgehv : ('a, 'b, 'c, 'd, 'e) t -> int`

Number of edges (hyperedge,vertex) in the hypergraph

`val size : ('a, 'b, 'c, 'd, 'e) t -> int * int * int * int`

`size graph` returns `(nbvertex,nbhedge,nbedgevh,nbedgehv)`

24.2.2 Information associated to vertives and edges

```
val attrvertex : ('a, 'b, 'c, 'd, 'e) t -> 'a -> 'c
```

`attrvertex graph vertex` returns the information associated to the vertex `vertex`

```
val attrhedge : ('a, 'b, 'c, 'd, 'e) t -> 'b -> 'd
```

`attrhedge graph hedge` returns the information associated to the hyperedge `hedge`

```
val info : ('a, 'b, 'c, 'd, 'e) t -> 'e
```

`info g` returns the user-information attached to the graph `g`

24.2.3 Membership tests

```
val is_vertex : ('a, 'b, 'c, 'd, 'e) t -> 'a -> bool
val is_hedge : ('a, 'b, 'c, 'd, 'e) t -> 'b -> bool
```

24.2.4 Successors and predecessors

```
val succhedge : ('a, 'b, 'c, 'd, 'e) t -> 'a -> 'b Sette.t
```

Successor hyperedges of a vertex

```
val predhedge : ('a, 'b, 'c, 'd, 'e) t -> 'a -> 'b Sette.t
```

Predecessor hyperedges of a vertex

```
val succvertex : ('a, 'b, 'c, 'd, 'e) t -> 'b -> 'a array
```

Successor vertices of an hyperedge

```
val predvertex : ('a, 'b, 'c, 'd, 'e) t -> 'b -> 'a array
```

Predecessor vertices of an hyperedge

```
val succ_vertex : ('a, 'b, 'c, 'd, 'e) t -> 'a -> 'a Sette.t
```

Successor vertices of a vertex by any hyperedge

```
val pred_vertex : ('a, 'b, 'c, 'd, 'e) t -> 'a -> 'a Sette.t
```

Predecessor vertices of a vertex by any hyperedge

24.2.5 Adding and removing elements

```
val add_vertex : ('a, 'b, 'c, 'd, 'e) t -> 'a -> 'c -> unit
```

Add a vertex

```
val add_hedge :
  ('a, 'b, 'c, 'd, 'e) t ->
  'b -> 'd -> pred:'a array -> succ:'a array -> unit
```

Add an hyperedge. The predecessor and successor vertices should already exist in the graph. Otherwise, a `Failure` exception is raised.

```
val replace_attrvertex : ('a, 'b, 'c, 'd, 'e) t -> 'a -> 'c -> unit
```

Change the attribute of an existing vertex

```
val replace_attrhedge : ('a, 'b, 'c, 'd, 'e) t -> 'b -> 'd -> unit
```

Change the attribute of an existing hyperedge

```
val remove_vertex : ('a, 'b, 'c, 'd, 'e) t -> 'a -> unit
```

Remove the vertex from the graph, as well as all related hyperedges.

```
val remove_hedge : ('a, 'b, 'c, 'd, 'e) t -> 'b -> unit
```

Remove the hyperedge from the graph.

24.2.6 Iterators

```
val iter_vertex :
  ('a, 'b, 'c, 'd, 'e) t ->
  ('a -> 'c -> pred:'b Sette.t -> succ:'b Sette.t -> unit) -> unit
```

Iterates the function `f vertex attrvertex succhedges predhedges` to all vertices of the graph. `succhedges` (resp. `predhedges`) is the set of successor (resp. predecessor) hyperedges of the vertex

```
val iter_hedge :
  ('a, 'b, 'c, 'd, 'e) t ->
  ('b -> 'd -> pred:'a array -> succ:'a array -> unit) -> unit
```

Iterates the function `f hedge attrhedge succvertices predvertices` to all hyperedges of the graph. `succvertices` (resp. `predvertices`) is the set of successor (resp. predecessor) vertices of the hyperedge

Below are the `fold` versions of the previous functions.

```
val fold_vertex :
  ('a, 'b, 'c, 'd, 'e) t ->
  ('a -> 'c -> pred:'b Sette.t -> succ:'b Sette.t -> 'f -> 'f) -> 'f -> 'f
val fold_hedge :
  ('a, 'b, 'c, 'd, 'e) t ->
  ('b -> 'd -> pred:'a array -> succ:'a array -> 'f -> 'f) -> 'f -> 'f
```

Below are the `map` versions of the previous functions.

```
val map :
  ('a, 'b, 'c, 'd, 'e) t ->
  ('a -> 'c -> 'f) ->
  ('b -> 'd -> 'g) -> ('e -> 'h) -> ('a, 'b, 'f, 'g, 'h) t
```

24.2.7 Copy and Transpose

```
val copy :
  ('a -> 'b -> 'c) ->
  ('d -> 'e -> 'f) ->
  ('g -> 'h) ->
  ('a, 'd, 'b, 'e, 'g) t -> ('a, 'd, 'c, 'f, 'h) t
```

Copy an hypergraph, using the given functions to duplicate the attributes associated to the elements of the graph. The vertex and hedge identifiers are copied using the identity function.

```
val transpose :
  ('a -> 'b -> 'c) ->
  ('d -> 'e -> 'f) ->
  ('g -> 'h) ->
  ('a, 'd, 'b, 'e, 'g) t -> ('a, 'd, 'c, 'f, 'h) t
```

Similar to `copy`, but hyperedges are reversed: successor vertices and predecessor vertices are exchanged.

24.2.8 Algorithms

```
val min : ('a, 'b, 'c, 'd, 'e) t -> 'a Sette.t
```

Return the set of vertices without predecessor hyperedges

```
val max : ('a, 'b, 'c, 'd, 'e) t -> 'a Sette.t
```

Return the set of vertices without successor hyperedges

24.2.8.1 Topological sort

```
val topological_sort :
  ?priority:'a priority ->
  ('b, 'a, 'c, 'd, 'e) t -> 'b -> 'b list
```

Topological sort of the vertices of the hypergraph starting from a root vertex. The graph supposed to be acyclic. Any hyperedge linking two vertices (which are resp. predecessor and successor) induces a dependency. The result contains only vertices reachable from the given root vertex. If the dependencies are cyclic, the result is meaningless.

```
val topological_sort_multi :
  'a ->
  'b ->
  ?priority:'b priority ->
  ('a, 'b, 'c, 'd, 'e) t -> 'a Sette.t -> 'a list
```

Topological sort from a set of root vertices. The two first arguments are supposed to be yet unused vertex and hyperedge identifier.

24.2.8.2 Reachability and coreachability

The variants of the basic functions are similar to the variants described above.

```
val reachable :
  ?filter:('a -> bool) ->
  ('b, 'a, 'c, 'd, 'e) t -> 'b -> 'b Sette.t * 'a Sette.t
```

Returns the set of vertices and hyperedges that are *NOT* reachable from the given root vertex. Any dependency in the sense described above is taken into account to define the reachability relation. For instance, if one of the predecessor vertex of an hyperedge is reachable, the hyperedge is considered as reachable.

```
val reachable_multi :
  'a ->
  'b ->
  ?filter:('b -> bool) ->
  ('a, 'b, 'c, 'd, 'e) t -> 'a Sette.t -> 'a Sette.t * 'b Sette.t
```

24.2.8.3 Strongly Connected Components and SubComponents

```
val cfc :
  ?priority:'a priority ->
  ('b, 'a, 'c, 'd, 'e) t -> 'b -> 'b list list
```

Decomposition of the graph into Strongly Connected Components,

`cfc graph vertex` returns a decomposition of the graph. The exploration is done from the initial vertex `vertex`, and only reachable vertices are included in the result. The result has the structure `[comp1 comp2 comp3 ...]` where each component is defined by a list of vertices. The ordering of component correspond to a linearization of the partial order between the components.

```
val cfc_multi :
  'a ->
  'b ->
  ?priority:'b priority ->
  ('a, 'b, 'c, 'd, 'e) t -> 'a Sette.t -> 'a list list
```

idem, but from several initial vertices.

`cfc dummy_vertex dummy_hedge graph setvertices` returns a decomposition of the graph, explored from the set of initial vertices `setvertices`. `dummy_vertex` and `dummy_hedge` are resp. unused vertex and hyperedge identifiers.

```
val scfc :
  ?priority:'a priority ->
  ('b, 'a, 'c, 'd, 'e) t -> 'b -> 'b Ilist.t
```

Decomposition of the graph into Strongly Connected Sub-Components,

`scfc graph vertex` returns a decomposition of the graph. The exploration is done from the initial vertex `vertex`, and only reachable vertices are included in the result. The result has the structure `[comp1 comp2 comp3 ...]` where each component is in turn decomposed into components.

```
val scfc_multi :
  'a ->
  'b ->
  ?priority:'b priority ->
  ('a, 'b, 'c, 'd, 'e) t -> 'a Sette.t -> 'a Ilist.t
```

idem, but from several initial vertices.

24.2.9 Printing

```
val print :
  (Format.formatter -> 'a -> unit) ->
  (Format.formatter -> 'b -> unit) ->
  (Format.formatter -> 'c -> unit) ->
  (Format.formatter -> 'd -> unit) ->
  (Format.formatter -> 'e -> unit) ->
  Format.formatter -> ('a, 'b, 'c, 'd, 'e) t -> unit
```

Print a graph in textual format on the given formatter, using the given functions to resp. print: vertices (`'a`), hedges (`'b`), vertex attributes (`'c`), hedge attributes (`'d`), and the user information (`'e`).

```
val print_dot :
  ?style:string ->
  ?titlestyle:string ->
  ?vertexstyle:string ->
  ?hedgestyle:string ->
  ?fvertexstyle:('a -> string) ->
  ?fhedgestyle:('b -> string) ->
  ?title:string ->
  (Format.formatter -> 'a -> unit) ->
  (Format.formatter -> 'b -> unit) ->
  (Format.formatter -> 'a -> 'c -> unit) ->
  (Format.formatter -> 'b -> 'd -> unit) ->
  Format.formatter -> ('a, 'b, 'c, 'd, 'e) t -> unit
```

Output the graph in DOT format on the given formatter, using the given functions to resp print:

- vertex identifiers (in the DOT file)
- hedge identifiers (in the DOT file).
  BE CAUTIOUS. as the DOT files vertices and hedges are actually nodes, the user should take care to avoid name conflicts between vertex and hedge names.
- vertex attributes.
  BE CAUTIOUS: the output of the function will be enclosed bewteen quotes. If ever the output contains line break, or other special characters, it should be escaped. A possible scheme to do this is to first output to `Format.str_formatter` with a standard printing function, then to escape the resulting string and to output the result. This gives something like:
  `print_attrvertex Format.str_formatter vertex attr;`
  `Format.pp_print_string fmt (String.escaped (Format.flush_str_formatter ()));`.
  Concerning the escape function, you may use `String.escaped`, which will produce center justified line breaks, or `Print.escaped` which allows also to choose between center, left and right justified lines.
- hedge atributes (same comment as for vertex attributes).

The optional arguments allows to customize the style. The default setting corresponds to:

```
print_dot ~style="ranksep=0.1; size=\"7,10\";"
~titlestyle="shape=ellipse,style=bold,style=filled,fontsize=20"
~vertexstyle="shape=box,fontsize=12" ~hedgestyle="shape=ellipse,fontsize=12"
~title="" ....
```

24.3 Parameter module for the functor version

```
module type T =
  sig
```

`type vertex`

Type of vertex identifiers

`type hedge`

Type of hyperedge identifiers

`val vertex_dummy : vertex`

A dummy (never used) value for vertex identifiers (used for the functions `XXX_multi`)

`val hedge_dummy : hedge`

A dummy (never used) value for hyperedge identifiers (used for the functions `XXX_multi`)

```
module SetV :
Sette.S  with type elt=vertex
```

Set module for vertices

```
module SetH :
Sette.S  with type elt=hedge
```

Set module for hyperedges

```
module HashV :
Hashhe.S  with type key=vertex
```

Hash module with vertices as keys

```
module HashH :
Hashhe.S  with type key=hedge
```

Hash module with hyperedges as keys

```
end
```

24.4 Signature of the functor version

All functions have the same signature as the polymorphic version, except the functions `XXX_multi` which does not need any more a dummy value of type `vertex` (resp. `hedge`).

```
module type S =
  sig
    type vertex
    type hedge
    val vertex_dummy : vertex
    val hedge_dummy : hedge
    module SetV :
    Sette.S  with type elt=vertex
    module SetH :
    Sette.S  with type elt=hedge
    module HashV :
    Hashhe.S  with type key=vertex
    module HashH :
    Hashhe.S  with type key=hedge
    val stdcompare : (vertex, hedge) SHGraph.compare
    type ('a, 'b, 'c) t
```

Type of hypergraphs, where

- 'a : information associated to vertices
- 'b : information associated to hedges
- 'c : user-information associated to an hypergraph

```
    val create : int -> 'a -> ('b, 'c, 'a) t
    val clear : ('a, 'b, 'c) t -> unit
    val is_empty : ('a, 'b, 'c) t -> bool
```

24.4.1 Statistics

```
    val size_vertex : ('a, 'b, 'c) t -> int
    val size_hedge : ('a, 'b, 'c) t -> int
    val size_edgevh : ('a, 'b, 'c) t -> int
    val size_edgehv : ('a, 'b, 'c) t -> int
    val size : ('a, 'b, 'c) t -> int * int * int * int
```

24.4.2 Information associated to vertives and edges

```
val attrvertex : ('a, 'b, 'c) t -> vertex -> 'a
```

```
val attrhedge : ('a, 'b, 'c) t -> hedge -> 'b
```

```
val info : ('a, 'b, 'c) t -> 'c
```

24.4.3 Membership tests

```
val is_vertex : ('a, 'b, 'c) t -> vertex -> bool
```

```
val is_hedge : ('a, 'b, 'c) t -> hedge -> bool
```

24.4.4 Successors and predecessors

```
val succhedge : ('a, 'b, 'c) t -> vertex -> SetH.t
```

```
val predhedge : ('a, 'b, 'c) t -> vertex -> SetH.t
```

```
val succvertex : ('a, 'b, 'c) t -> hedge -> vertex array
```

```
val predvertex : ('a, 'b, 'c) t -> hedge -> vertex array
```

```
val succ_vertex : ('a, 'b, 'c) t -> vertex -> SetV.t
```

```
val pred_vertex : ('a, 'b, 'c) t -> vertex -> SetV.t
```

24.4.5 Adding and removing elements

```
val add_vertex : ('a, 'b, 'c) t -> vertex -> 'a -> unit
```

```
val add_hedge :
  ('a, 'b, 'c) t ->
  hedge ->
  'b -> pred:vertex array -> succ:vertex array -> unit
```

```
val replace_attrvertex : ('a, 'b, 'c) t -> vertex -> 'a -> unit
```

```
val replace_attrhedge : ('a, 'b, 'c) t -> hedge -> 'b -> unit
```

```
val remove_vertex : ('a, 'b, 'c) t -> vertex -> unit
```

```
val remove_hedge : ('a, 'b, 'c) t -> hedge -> unit
```

24.4.6 Iterators

```
val iter_vertex :
  ('a, 'b, 'c) t ->
  (vertex ->
   'a -> pred:SetH.t -> succ:SetH.t -> unit) ->
  unit
```

```
val iter_hedge :
  ('a, 'b, 'c) t ->
  (hedge ->
   'b -> pred:vertex array -> succ:vertex array -> unit) ->
  unit
```

```
val fold_vertex :
  ('a, 'b, 'c) t ->
  (vertex ->
   'a -> pred:SetH.t -> succ:SetH.t -> 'd -> 'd) ->
  'd -> 'd
```

```
val fold_hedge :
  ('a, 'b, 'c) t ->
  (hedge ->
   'b -> pred:vertex array -> succ:vertex array -> 'd -> 'd) ->
  'd -> 'd
```

24.4.7 Copy and Transpose

```
val map :
  ('a, 'b, 'c) t ->
  (vertex -> 'a -> 'd) ->
  (hedge -> 'b -> 'e) -> ('c -> 'f) -> ('d, 'e, 'f) t
val copy :
  (vertex -> 'a -> 'b) ->
  (hedge -> 'c -> 'd) ->
  ('e -> 'f) -> ('a, 'c, 'e) t -> ('b, 'd, 'f) t
val transpose :
  (vertex -> 'a -> 'b) ->
  (hedge -> 'c -> 'd) ->
  ('e -> 'f) -> ('a, 'c, 'e) t -> ('b, 'd, 'f) t
```

24.4.8 Algorithms

```
val min : ('a, 'b, 'c) t -> SetV.t
val max : ('a, 'b, 'c) t -> SetV.t
val topological_sort :
  ?priority:hedge SHGraph.priority ->
  ('a, 'b, 'c) t -> vertex -> vertex list
val topological_sort_multi :
  ?priority:hedge SHGraph.priority ->
  ('a, 'b, 'c) t -> SetV.t -> vertex list
val reachable :
  ?filter:(hedge -> bool) ->
  ('a, 'b, 'c) t ->
  vertex -> SetV.t * SetH.t
val reachable_multi :
  ?filter:(hedge -> bool) ->
  ('a, 'b, 'c) t ->
  SetV.t -> SetV.t * SetH.t
val cfc :
  ?priority:hedge SHGraph.priority ->
  ('a, 'b, 'c) t -> vertex -> vertex list list
val cfc_multi :
  ?priority:hedge SHGraph.priority ->
  ('a, 'b, 'c) t -> SetV.t -> vertex list list
val scfc :
  ?priority:hedge SHGraph.priority ->
  ('a, 'b, 'c) t -> vertex -> vertex Ilist.t
val scfc_multi :
  ?priority:hedge SHGraph.priority ->
  ('a, 'b, 'c) t -> SetV.t -> vertex Ilist.t
```

24.4.9 Printing

```
    val print :
      (Format.formatter -> vertex -> unit) ->
      (Format.formatter -> hedge -> unit) ->
      (Format.formatter -> 'a -> unit) ->
      (Format.formatter -> 'b -> unit) ->
      (Format.formatter -> 'c -> unit) ->
      Format.formatter -> ('a, 'b, 'c) t -> unit
    val print_dot :
      ?style:string ->
      ?titlestyle:string ->
      ?vertexstyle:string ->
      ?hedgestyle:string ->
      ?fvertexstyle:(vertex -> string) ->
      ?fhedgestyle:(hedge -> string) ->
      ?title:string ->
      (Format.formatter -> vertex -> unit) ->
      (Format.formatter -> hedge -> unit) ->
      (Format.formatter -> vertex -> 'a -> unit) ->
      (Format.formatter -> hedge -> 'b -> unit) ->
      Format.formatter -> ('a, 'b, 'c) t -> unit
  end
```

24.5 Functor

```
module Make :
functor (T : T) -> S  with type vertex=T.vertex and type hedge=T.hedge and module SetV=T.SetV
and module SetH=T.SetH and module HashV=T.HashV and module HashH=T.HashH
```

24.6 Compare interface

```
module Compare :
  sig

    val attrvertex :
      ('a, 'b) SHGraph.compare -> ('a, 'c, 'd, 'e, 'f) SHGraph.graph -> 'a -> 'd
    val attrhedge :
      ('a, 'b) SHGraph.compare -> ('c, 'b, 'd, 'e, 'f) SHGraph.graph -> 'b -> 'e
    val is_vertex :
      ('a, 'b) SHGraph.compare -> ('a, 'c, 'd, 'e, 'f) SHGraph.graph -> 'a -> bool
    val is_hedge :
      ('a, 'b) SHGraph.compare -> ('c, 'b, 'd, 'e, 'f) SHGraph.graph -> 'b -> bool
    val succhedge :
      ('a, 'b) SHGraph.compare ->
      ('a, 'c, 'd, 'e, 'f) SHGraph.graph -> 'a -> 'c Sette.t
    val predhedge :
      ('a, 'b) SHGraph.compare ->
      ('a, 'c, 'd, 'e, 'f) SHGraph.graph -> 'a -> 'c Sette.t
    val succvertex :
      ('a, 'b) SHGraph.compare ->
      ('c, 'b, 'd, 'e, 'f) SHGraph.graph -> 'b -> 'c array
    val predvertex :
      ('a, 'b) SHGraph.compare ->
      ('c, 'b, 'd, 'e, 'f) SHGraph.graph -> 'b -> 'c array
    val succ_vertex :
      ('a, 'b) SHGraph.compare ->
      ('a, 'b, 'c, 'd, 'e) SHGraph.graph -> 'a -> 'a Sette.t
    val pred_vertex :
      ('a, 'b) SHGraph.compare ->
      ('a, 'b, 'c, 'd, 'e) SHGraph.graph -> 'a -> 'a Sette.t
    val add_vertex :
      ('a, 'b) SHGraph.compare ->
      ('a, 'c, 'd, 'e, 'f) SHGraph.graph -> 'a -> 'd -> unit
    val add_hedge :
      ('a, 'b) SHGraph.compare ->
      ('a, 'b, 'c, 'd, 'e) SHGraph.graph ->
      'b -> 'd -> pred:'a array -> succ:'a array -> unit
    val replace_attrvertex :
      ('a, 'b) SHGraph.compare ->
      ('a, 'c, 'd, 'e, 'f) SHGraph.graph -> 'a -> 'd -> unit
    val replace_attrhedge :
      ('a, 'b) SHGraph.compare ->
      ('a, 'b, 'c, 'd, 'e) SHGraph.graph -> 'b -> 'd -> unit
    val remove_hedge :
      ('a, 'b) SHGraph.compare -> ('a, 'b, 'c, 'd, 'e) SHGraph.graph -> 'b -> unit
    val remove_vertex :
      ('a, 'b) SHGraph.compare -> ('a, 'b, 'c, 'd, 'e) SHGraph.graph -> 'a -> unit
    val topological_sort :
      ('a, 'b) SHGraph.compare ->
      ?priority:'b SHGraph.priority ->
      ('a, 'b, 'c, 'd, 'e) SHGraph.t -> 'a -> 'a list
    val topological_sort_multi :
      ('a, 'b) SHGraph.compare ->
      'a ->
      'b ->
      ?priority:'b SHGraph.priority ->
      ('a, 'b, 'c, 'd, 'e) SHGraph.t -> 'a Sette.t -> 'a list
    val reachable :
      ('a, 'b) SHGraph.compare ->
      ?filter:('b -> bool) ->
      ('a, 'b, 'c, 'd, 'e) SHGraph.t -> 'a -> 'a Sette.t * 'b Sette.t
    val reachable_multi :
      ('a, 'b) SHGraph.compare ->
      'a ->
      'b ->
      ?filter:('b -> bool) ->
      ('a, 'b, 'c, 'd, 'e) SHGraph.t -> 'a Sette.t -> 'a Sette.t * 'b Sette.t
    val cfc :
      ('a, 'b) SHGraph.compare ->
      ?priority:'b SHGraph.priority ->
      ('a, 'b, 'c, 'd, 'e) SHGraph.graph -> 'a -> 'a list list
    val cfc_multi :
      ('a, 'b) SHGraph.compare ->
      ?priority:'b SHGraph.priority ->
      'a -> 'b -> ('a, 'b, 'c, 'd, 'e) SHGraph.graph -> 'a Sette.t -> 'a list list
    val scfc :
      ('a, 'b) SHGraph.compare ->
      ?priority:'b SHGraph.priority ->
      ('a, 'b, 'c, 'd, 'e) SHGraph.graph -> 'a -> 'a Ilist.t
    val scfc_multi :
      ('a, 'b) SHGraph.compare ->
      'a ->
      'b ->
      ?priority:'b SHGraph.priority ->
      ('a, 'b, 'c, 'd, 'e) SHGraph.graph -> 'a Sette.t -> 'a Ilist.t
    val print :
      ('a, 'b) SHGraph.compare ->
      (Format.formatter -> 'a -> unit) ->
      (Format.formatter -> 'b -> unit) ->
      (Format.formatter -> 'c -> unit) ->
      (Format.formatter -> 'd -> unit) ->
      (Format.formatter -> 'e -> unit) ->
      Format.formatter -> ('a, 'b, 'c, 'd, 'e) SHGraph.graph -> unit
    val min :
      ('a, 'b) SHGraph.compare -> ('a, 'c, 'd, 'e, 'f) SHGraph.graph -> 'a Sette.t
    val max :
      ('a, 'b) SHGraph.compare -> ('a, 'c, 'd, 'e, 'f) SHGraph.graph -> 'a Sette.t
 end
```

Chapter 25

Module PSHGraph : Oriented hypergraphs, parametrized polymorphic version.

Same interface as SHGraph[24], but each graph stores its comparison functions.
See the comment about comparison functions in the module PSette[3].

```
type 'a priority = 'a SHGraph.priority =
  | Filter of ('a -> bool)
  | Priority of ('a -> int)
```

```
type ('a, 'b) compare = ('a, 'b) SHGraph.compare = {
  hashv : 'a Hashhe.compare ;
  hashh : 'b Hashhe.compare ;
  comparev : 'a -> 'a -> int ;
  compareh : 'b -> 'b -> int ;
}
```

```
type ('a, 'b, 'c, 'd, 'e) t = {
  compare : ('a, 'b) compare ;
  mutable graph : ('a, 'b, 'c, 'd, 'e) SHGraph.t ;
}
```

```
val stdcompare : ('a, 'b) compare
```

```
val make :
  ('a, 'b) compare ->
  ('a, 'b, 'c, 'd, 'e) SHGraph.t -> ('a, 'b, 'c, 'd, 'e) t
```

```
val create_compare : ('a, 'b) compare -> int -> 'c -> ('a, 'b, 'd, 'e, 'c) t
```

```
val create : ('a, 'b) compare -> int -> 'c -> ('a, 'b, 'd, 'e, 'c) t
```

```
val clear : ('a, 'b, 'c, 'd, 'e) t -> unit
```

```
val size_vertex : ('a, 'b, 'c, 'd, 'e) t -> int
```

```
val size_hedge : ('a, 'b, 'c, 'd, 'e) t -> int
```

```
val size_edgevh : ('a, 'b, 'c, 'd, 'e) t -> int
```

```
val size_edgehv : ('a, 'b, 'c, 'd, 'e) t -> int
```

```
val size : ('a, 'b, 'c, 'd, 'e) t -> int * int * int * int
```

```
val attrvertex : ('a, 'b, 'c, 'd, 'e) t -> 'a -> 'c
```

```
val attrhedge : ('a, 'b, 'c, 'd, 'e) t -> 'b -> 'd
```

```
val info : ('a, 'b, 'c, 'd, 'e) t -> 'e
val is_vertex : ('a, 'b, 'c, 'd, 'e) t -> 'a -> bool
```

```
val is_hedge : ('a, 'b, 'c, 'd, 'e) t -> 'b -> bool
```

```
val is_empty : ('a, 'b, 'c, 'd, 'e) t -> bool
```

```
val succhedge : ('a, 'b, 'c, 'd, 'e) t -> 'a -> 'b PSette.t
```

```
val predhedge : ('a, 'b, 'c, 'd, 'e) t -> 'a -> 'b PSette.t
```

```
val succvertex : ('a, 'b, 'c, 'd, 'e) t -> 'b -> 'a array
```

```
val predvertex : ('a, 'b, 'c, 'd, 'e) t -> 'b -> 'a array
```

```
val succ_vertex : ('a, 'b, 'c, 'd, 'e) t -> 'a -> 'a PSette.t
```

```
val pred_vertex : ('a, 'b, 'c, 'd, 'e) t -> 'a -> 'a PSette.t
```

```
val add_vertex : ('a, 'b, 'c, 'd, 'e) t -> 'a -> 'c -> unit
```

```
val add_hedge :
  ('a, 'b, 'c, 'd, 'e) t ->
  'b -> 'd -> pred:'a array -> succ:'a array -> unit
```

```
val replace_attrvertex : ('a, 'b, 'c, 'd, 'e) t -> 'a -> 'c -> unit
```

```
val replace_attrhedge : ('a, 'b, 'c, 'd, 'e) t -> 'b -> 'd -> unit
```

```
val remove_hedge : ('a, 'b, 'c, 'd, 'e) t -> 'b -> unit
```

```
val remove_vertex : ('a, 'b, 'c, 'd, 'e) t -> 'a -> unit
```

```
val iter_vertex :
  ('a, 'b, 'c, 'd, 'e) t ->
  ('a -> 'c -> pred:'b PSette.t -> succ:'b PSette.t -> unit) -> unit
```

```
val fold_vertex :
  ('a, 'b, 'c, 'd, 'e) t ->
  ('a -> 'c -> pred:'b PSette.t -> succ:'b PSette.t -> 'f -> 'f) -> 'f -> 'f
```

```
val iter_hedge :
  ('a, 'b, 'c, 'd, 'e) t ->
  ('b -> 'd -> pred:'a array -> succ:'a array -> unit) -> unit
```

```
val fold_hedge :
  ('a, 'b, 'c, 'd, 'e) t ->
  ('b -> 'd -> pred:'a array -> succ:'a array -> 'f -> 'f) -> 'f -> 'f
```

```
val map :
  ('a, 'b, 'c, 'd, 'e) t ->
  ('a -> 'c -> 'f) ->
  ('b -> 'd -> 'g) -> ('e -> 'h) -> ('a, 'b, 'f, 'g, 'h) t
```

```
val copy :
  ('a -> 'b -> 'c) ->
  ('d -> 'e -> 'f) ->
  ('g -> 'h) ->
  ('a, 'd, 'b, 'e, 'g) t -> ('a, 'd, 'c, 'f, 'h) t
```

```
val transpose :
  ('a -> 'b -> 'c) ->
  ('d -> 'e -> 'f) ->
  ('g -> 'h) ->
  ('a, 'd, 'b, 'e, 'g) t -> ('a, 'd, 'c, 'f, 'h) t
```

```
val topological_sort :
  ?priority:'a priority ->
  ('b, 'a, 'c, 'd, 'e) t -> 'b -> 'b list
```

```
val topological_sort_multi :
  'a ->
  'b ->
  ?priority:'b priority ->
  ('a, 'b, 'c, 'd, 'e) t -> 'a PSette.t -> 'a list
val reachable :
  ?filter:('a -> bool) ->
  ('b, 'a, 'c, 'd, 'e) t -> 'b -> 'b PSette.t * 'a PSette.t
val reachable_multi :
  'a ->
  'b ->
  ?filter:('b -> bool) ->
  ('a, 'b, 'c, 'd, 'e) t -> 'a PSette.t -> 'a PSette.t * 'b PSette.t
val cfc :
  ?priority:'a priority ->
  ('b, 'a, 'c, 'd, 'e) t -> 'b -> 'b list list
val cfc_multi :
  'a ->
  'b ->
  ?priority:'b priority ->
  ('a, 'b, 'c, 'd, 'e) t -> 'a PSette.t -> 'a list list
val scfc :
  ?priority:'a priority ->
  ('b, 'a, 'c, 'd, 'e) t -> 'b -> 'b Ilist.t
val scfc_multi :
  'a ->
  'b ->
  ?priority:'b priority ->
  ('a, 'b, 'c, 'd, 'e) t -> 'a PSette.t -> 'a Ilist.t
val print :
  (Format.formatter -> 'a -> unit) ->
  (Format.formatter -> 'b -> unit) ->
  (Format.formatter -> 'c -> unit) ->
  (Format.formatter -> 'd -> unit) ->
  (Format.formatter -> 'e -> unit) ->
  Format.formatter -> ('a, 'b, 'c, 'd, 'e) t -> unit
val print_dot :
  ?style:string ->
  ?titlestyle:string ->
  ?vertexstyle:string ->
  ?hedgestyle:string ->
  ?fvertexstyle:('a -> string) ->
  ?fhedgestyle:('b -> string) ->
  ?title:string ->
  (Format.formatter -> 'a -> unit) ->
  (Format.formatter -> 'b -> unit) ->
  (Format.formatter -> 'a -> 'c -> unit) ->
  (Format.formatter -> 'b -> 'd -> unit) ->
  Format.formatter -> ('a, 'b, 'c, 'd, 'e) t -> unit
val min : ('a, 'b, 'c, 'd, 'e) t -> 'a PSette.t
val max : ('a, 'b, 'c, 'd, 'e) t -> 'a PSette.t
```

Chapter 26

Module `Symbol` : Symbol table, for string management

`type t`

Type of symbols (actually integers)

`val add : string -> t`

Returns the symbol associated to the given string, after having possibly registered the string if it wasn't (in this case, the symbol is fresh symbol).

`val exists : string -> bool`

Is the string already registered ?

`val of_string : string -> t`

Returns the *existing* symbol associated to the (registered) string. Raises `Not_found` otherwise.

`val to_string : t -> string`

Returns the string represented by the symbol.

`val print : Format.formatter -> t -> unit`

Prints the symbol (its associated string).

`val equal : t -> t -> bool`

Equality test

`val compare : t -> t -> int`

Comparison (do not correspond at all to alphabetic order, depend on the registration order of names in the module)

```
module HashedType :
Hashtbl.HashedType  with type t=t
```

To use hashtables on type `t`

```
module OrderedType :
Set.OrderedType  with type t=t
```

To use sets or maps on type `t`

```
module Hash :
Hashtbl.S  with type key=t
```

Hashtables on type `t`

```
module Set :
Sette.S  with type elt=t
```

Sets on type `t`

```
module Map :
Mappe.S  with type key=t and module Setkey=Set
```

Maps on type `t`

```
val print_set : Format.formatter -> Set.t -> unit
```

Prints sets of symbols.

```
val print_hash :
  (Format.formatter -> 'a -> unit) ->
  Format.formatter -> 'a Hash.t -> unit
```

Prints hashtables on symbols.

```
val print_map :
  (Format.formatter -> 'a -> unit) ->
  Format.formatter -> 'a Map.t -> unit
```

Prints maps on symbols.

Chapter 27

Module `Union` : Union-find Abstract Data Types

`type 'a t`

The type of the data structure storing set membership (the universe)

`val create : int -> 'a t`

Create a new universe set

`val add : 'a t -> 'a -> unit`

Add an element to the universe (initially belonging to its singleton)

`val find : 'a t -> 'a -> 'a`

`val union : 'a t -> 'a -> 'a -> 'a`

Computes the union of two sets and returns the resulting set

`val extract : 'a t -> 'a list list`

Extract the list of sets

Chapter 28

Module `Statistic` : Standard statistics functions

`val mean : float array -> float`

Returns the mean of the array.

`val variance : float -> float array -> float`

Given the mean, returns the variance of the array.

`val std_deviation : float -> float array -> float`

Given the mean, returns the standard deviation of the array.

Chapter 29

Module Time : Small module to compute the duration of computations

`val wrap_duration : float Pervasives.ref -> (unit -> 'a) -> 'a`

`wrap_duration duration f` executes the function `f` and stores into `!duration` the time spent in `f`, in seconds. If `f` raises an exception, the exception is transmitted and the computed duration is still valid.

`val wrap_duration_add : float Pervasives.ref -> (unit -> 'a) -> 'a`

Similar to `wrap_duration`, but here the time spent in `f` is added to the value `!duration`.

Chapter 30

Module Rational : Rational numbers

```
val gcd : int -> int -> int
type t = {
  num : int ;
  den : int ;
}
val make : int -> int -> t
val inv : t -> t
val neg : t -> t
val add : t -> t -> t
val sub : t -> t -> t
val mul : t -> t -> t
val div : t -> t -> t
val to_string : t -> string
val print : Format.formatter -> t -> unit
```

Chapter 31

Module Parse : Linking lexer and parser function, with (basic) error messages

```
exception Lex_error
```

```
val lex_eol : Lexing.lexbuf -> unit
```

Function to call on line returns in lexer

```
val parse_lexbuf :
  lexer:(Lexing.lexbuf -> 'a) ->
  parser:((Lexing.lexbuf -> 'a) -> Lexing.lexbuf -> 'b) -> Lexing.lexbuf -> 'b
```

Takes as input a lexer, a parser compatible with the lexer (ie, they share the same type for tokens), and a `lexbuf`,

Returns the AST built by the parser.

```
val parse_string :
  lexer:(Lexing.lexbuf -> 'a) ->
  parser:((Lexing.lexbuf -> 'a) -> Lexing.lexbuf -> 'b) -> string -> 'b
```

Same as before, but create itself a `lexbuf` from the given string.

```
val parse_file :
  lexer:(Lexing.lexbuf -> 'a) ->
  parser:((Lexing.lexbuf -> 'a) -> Lexing.lexbuf -> 'b) -> string -> 'b
```

Same as before, but create itself a `lexbuf` from the given filename.

Index