How files correspond to modules.

- Compiling name.mli corresponds to interpreting
  module **type** Name = sig
  contents of name.mli
  end
- Compiling name.ml if name.mli exists corresponds to
  module Name : Name = sig
  contents of name.ml
  end
- Compiling a file name.ml if name.mli is missing means
  module Name = struct
  contents of name.ml
  end

Example

Compiling

inc.mli

```
val inc : int -> int
```

inc.ml

```
let inc x = x+1
```

main.ml

```
open Inc;;
print_int (inc 3);;
print_newline();;
```

Example

is the same as interpreting

```
module type Inc = sig
  val inc : int -> int
end;;
module Inc : Inc = struct
  let inc x = x+1
end;;
open Inc;;
print_int (inc 3);;
print_newline();;
```

Small Problem

Files cannot be functors , however

- module types can contain module types

```
module type M = sig
  module type N = sig
   val id : 'a -> 'a
  end
end;;
```

- functors and modules can contain modules and functors

```
module M = struct
  module N = struct
    let id x = x
  end
end;;
```

How To Compile

type	to produce
ocamlc -c inc.mli	inc.cmi
ocamlc -c inc.ml	inc.cmo
ocamlc -c main.ml	main.cmo
ocamlc -o main inc.cmo main.cmo	main
./main	4

Details

- Before compiling a file containing

  ```
  open Name
  ```

  name.cmi must have been generated:
  - by compiling name.mli if it exists
  - by compiling name.ml otherwise
- The order of linking matters:
  if F opens G then g.cmo must be to the left of f.cmo
- The main module is nothing special:
  - any module can contain initialization code/main code
  - code in modules is executed in the order they were linked.

Comparison

- The interpreter
  - Can read one file and/or standard input.
  - Gives pretty printing functions for free.
- The (byte-code) compiler
  - Compiles as many files as needed separate or together.
  - Links objects into binaries.
  - Makes the user responsible for pretty printing.
- The `ocamltry` script
  - Collects several .mli and .ml files in a single file.
  - Starts the interpreter preloaded with that file.
  - Shows internal details if the .mli is omitted.

Generating things

- $seq\ n\ k = [n; n+1; \cdots; k]$
- $sublists\ xs$: the list of all sublists of xs:

$$sublists\ [1;2;3] = [[]; [1]; [2]; [3]; [1;2]; [1;3]; [2;3]; [1;2;3]]$$

Convention: for a list of lists by default

- the order of the returned list is irrelevant
- the multiplicity of the elements counts
- the order and multiplicity of the elements counts

Permutations

- *insert x xs*: list of lists obtained by inserting *x* into *xs*:

$$insert\ 2\ [1;3] = [[2;1;3];[1;2;3];[1;3;2]]$$

- *permute xs*: list of all possible permutations of *xs*:

$$permute\ [1;2;3] = [[1;2;3];[1;3;2];[2;1;3];\\ [2;3;1];[3;1;2];[3;2;1]]$$

Filtering

- Remember filter:

```
let rec filter p l = match l with
  | [] -> []
  | x::xs when p x -> x :: (filter p xs)
  | x::xs -> (filter p xs)
;;
```

- *sum_is n ns*: checks if the sum of *ns* is *n*.
- *len_is n xs*: checks if the length of *ns* is *n*.

Application

- *kakuro n k*: generates a list of all possible combinations of *k* single digit numbers, such that each number occurs at most once and the sum of the numbers is *n*.
- *possibles ll*: given a list of combinations (a combination is a list) it produces a list of possible values at each position.

  $$\mathit{possibles}\ [[1;2];[3;2];[1;3];[3;1]] = [[1;2];[1;2;3]]$$
- *verify lp l*: given a list of possible values per position and a list check if the list has a possible value at each position.

  $$\mathit{verify}\ [[1;2];[2;1]]\ [1;2] = \mathit{true}$$
  $$\mathit{verify}\ [[1;2];[2;1]]\ [1;3] = \mathit{false}$$