1. Anagrams. The following program finds all the anagrams of a given set of characters among a dictionary. Such a dictionary can be built by the following program given a list of files containing words (one per line).

2. The dictionary is implemented as a *trie*. It is a multi-branching tree, where branches are labelled with characters. Each node contains a boolean which says if the word corresponding to the path from the root belongs to the dictionary. The branches are implemented as maps from characters to dictionaries.

```
module Cmap = Map.Make(struct type t = char let compare = compare end)

type tree = Node of bool × tree Cmap.t

let empty = Node (false, Cmap.empty)
```

3. Insertion of a new word in the trie is just a matter of descending in the tree, taking the branches corresponding to the successive characters. Each time a branch does not exist, we continue the insertion in a new empty tree. When the insertion is done in the subtree, we update the branching to the new subtree.

```
let add t w =
  let n = String.length w in
  let rec addrec i (Node (b, m) as t) =
    if i = n then
      if b then t else Node (true,m)
    else
      let c = w.[i] in
      let br = try Cmap.find c m with Not_found → empty in
      let t′ = addrec (succ i) br in
      Node (b, Cmap.add c t′ m)
  in
  addrec 0 t
```

4. Even if it is not necessary, here is the function *mem* to tests whether a word belongs or not to the dictionary.

```
let mem t w =
  let n = String.length w in
  let rec look i (Node (b, m)) =
    if i = n then
      b
    else
      try look (succ i) (Cmap.find w.[i] m) with Not_found → false
  in
  look 0 t
```

5. The algorithm for anagrans is the following. We start from the root of the tree with all the initial characters. Then, for each *distinct* character c, we descend in the corresponding branch, and apply recursively the algorithm with *one occurrence* of c being removed. When the collection of characters is empty, we simply test the boolean at the current node. Whenever a branch is missing, we stop the exploration.

It appears that we need to deal with *multi-sets* of characters. Indeed, we have to keep the collection of characters which have not yet been examined, and it may contain repetitions.

6. Multi-sets of characters are implemented as maps from characters to positive integers. The operations of insertion and deletion are easily implemented. We also provide a function *ms_of_string* to build the multi-set corresponding to a given string.

```
let ms_add c m =
  try let n = Cmap.find c m in Cmap.add c (succ n) m
  with Not_found → Cmap.add c 1 m

let ms_remove c m =
  let n = Cmap.find c m in
  if n = 1 then Cmap.remove c m else Cmap.add c (pred n) m

let ms_of_string w =
  let n = String.length w in
  let rec add i = if i = n then Cmap.empty else ms_add w.[i] (add (succ i)) in
  add 0
```

7. Then implementing the above algorithm is rather easy. During the exploration, we keep three values: first, the current path *pref* from the root of the initial tree, in reverse order; secondly, the current node being examined, (b, m); and finally, the current multi-set of characters s.

```
let subset = ref true

let rec print_prefix = function
  | [] → ()
  | c :: l → print_prefix l; print_char c

let anagram d w =
  let rec traverse pref (Node (b, m)) s =
    if b ∧ (s = Cmap.empty ∨ !subset) then begin
      print_prefix pref; print_newline ()
    end;
    Cmap.iter
      (fun c _ →
          try traverse (c :: pref) (Cmap.find c m) (ms_remove c s)
```

```
      with Not_found → ()) s
  in
  traverse [] d (ms_of_string w)
```

8. Building the dictionary. The function *add_one_file* read all the words contained in file *file* and inserts them in the tree t. Then function *build_dict* constructs the whole dictionary by successively inserting the words for the given list of files.

```
let add_one_file t file =
  Printf.printf "Reading␣%s\n" file; flush stdout;
  let ch = open_in file in
  let rec read t =
    try let w = input_line ch in read (add t w) with End_of_file → t
  in
  let t′ = read t in close_in ch; t′

let build_dict = List.fold_left add_one_file empty
```

9. The following function *print_all* prints all the words of a given dictionary. Only used for checks (option `-a`).

```
let print_all d =
  let rec traverse pref (Node (b, m)) =
    if b then begin print_prefix pref; print_newline () end;
    Cmap.iter (fun c t → traverse (c :: pref) t) m
  in
  traverse [] d
```

10. The main program. It mainly provides two ways of invoking the program: first, the option `-b` will build the dictionary from the given files and put it in the file `"dict.out"`; secondly, the program invoked with a word on the command line will print all the anagrams for this word. Option `-e` specifies exact anagrams (i.e., with all characters used).

```
let output_dict d =
  let ch = open_out "dict.out" in output_value ch d; close_out ch

let input_dict () =
  let ch = open_in "dict.out" in let d = input_value ch in close_in ch; d

let usage () =
  prerr_endline "usage:";
  prerr_endline "␣␣anagram␣-b␣files";
  prerr_endline "␣␣anagram␣[-e]␣word"
```

```
let main () =
  match List.tl (Array.to_list Sys.argv) with
    | [] | "-h" :: _ → usage ()
    | "-a" :: _ → let d = input_dict () in print_all d
    | "-b" :: files → let d = build_dict files in output_dict d
    | "-e" :: w :: _ → subset := false; let d = input_dict () in anagram d w
    | w :: _ → let d = input_dict () in anagram d w

let _ = Printexc.catch main ()
```