The Trie Data Structure*

Topics

- Purely functional programming
- Algebraic data types
- Polymorphism

Introduction

In this series of exercises, we will learn to build tries. A trie is a form of persistent maps that associates list like keys to values.

For starters, we will build a degenerate version of the structure, immutable sets of words. Then we will associate values to words, and finally generalize to list keys instead of words.

Note: for the sake of simplicity, we will only use lists and polymorphic commparison for implementing this data structure, but the resulting complexity is not the best one could expect for tries.

Exercise 1 – dictionnaries

We will build sets of words (dictionnaries), with two primitives: `present` and `insert` (feel free to add others).

The (recursive) type for dictionnaries is:

```
type dict = Node of bool * (char * dict) list
```

It is a *n*-ary tree structure in which the `Node` constructor represents a node in the tree and the `list` the tagged edges to its children. Each node bears a `bool`, that indicates if the word whose letters are the tags of the edges from the root is a member or the set.

Bonus: here is a little program to test your code.

```
let rec main dict =
  match Str.(split (regexp "[␣\t]+") (read_line ())) with
  | [ "present" ; word ] ->
    print_endline (if present word dict then "Present." else  "Not␣present.") ;
    main dict
  | "insert" :: words ->
    let dict = List.fold_right insert words dict in
    print_endline "Added." ;
    main dict
  | "exit" ->
    print_endline "Bye."
  | _ ->
    print_endline "Unknown␣command." ;
    main dict
let () = main empty
```

Question 1.1 – Write an auxiliary function `explode` that turns a string into a list of characters. Use an internal tail recursive function (Bonus if you don't `List.rev`).

Question 1.2 – Write a value `empty` of type `dict`, representing the empty dictionnary (of which even the empty string is not a member).

Question 1.3 – Write `present` of type `string -> dict -> bool`, that tells if a word is present in the dictionnary.

Question 1.4 – Write `present` of type `string -> dict -> dict`, that returns a new dictionnary withan added word.

Exercise 2

We now generalize the dictionnary to a type `'a dictmap` that associates values of type `'a` to words.

Question 2.1 – Define the type `'a dictmap`.

Question 2.2 – Write `empty` of type `'a dictmap`.

Question 2.3 – Write `get` of type `string -> 'a dictmap -> 'a` that returns the values associated to a word or fails with `Not_found`.

Question 2.4 – Write `set` of type `string -> 'a -> 'a dictmap -> 'a dictmap` that initializes or updates the value associated to a word.

Question 2.5 – Update the test program to test your implementation of `string dictmap` .

Exercise 3

Question 3.1 – Same questions, but generalizes to a type `('a, 'b) trie` that associates values of type `'b` to keys of type `'a list`.

Question 3.2 – Update the test program to test your implementation of `(char, string) dictmap` .

The Trie Data Structure

OLUTIONS – SOLUTIONS – SOLUTIONS – SOLUTIO

Solution to question 1.1

```
let explode s =
  let rec loop acc i =
    if i < 0 then acc else loop (String.get s i :: acc) (pred i) in
  loop [] (String.length s - 1)
```

Solution to question 1.2

```
let empty : dict = Node (false, [])
```

Solution to question 1.3

```
let present : string -> dict -> bool = fun s dict ->
  let rec descend = function
    | [], Node (present, _) -> present
    | _ :: _, Node (_, []) -> false
    | c :: cs as s, Node (present, (subc, subdict) :: rest) ->
      if c = subc then
        descend (cs, subdict)
      else
        descend (s, Node (present, rest)) in
  descend (explode s, dict)
```

Solution to question 1.4

```
let insert : string -> dict -> dict = fun s dict ->
  let rec descend = function
    | [], Node (_, subs) -> Node (true, subs)
    | c :: cs, Node (present, []) ->
      Node (present, [ (c, descend (cs, empty)) ])
    | c :: cs as s, Node (present, (subc, subdict) :: rest) ->
      if c = subc then
        Node (present, (subc, descend (cs, subdict)) :: rest)
      else
        let Node (_, nrest) =  descend (s, Node (present, rest)) in
        Node (present, (subc, subdict) :: nrest)
  in
  descend (explode s, dict)
```

Solution to question 2.1

```
type 'a dictmap = Node of 'a option * (char * 'a dictmap) list
```

Solution to question 2.2

```
let empty : 'a dictmap = Node (None, [])
```

Solution to question 2.3

```
let get : string -> 'a dictmap -> 'a = fun s dict ->
  let rec descend = function
    | [], Node (Some value, _) -> value
    | [], Node (None, _) -> raise Not_found
    | _ :: _, Node (_, []) -> raise Not_found
    | c :: cs as s, Node (present, (subc, subdict) :: rest) ->
      if c = subc then
        descend (cs, subdict)
      else
        descend (s, Node (present, rest)) in
  descend (explode s, dict)
```

Solution to question 2.4

```
let set : string -> 'a -> 'a dictmap -> 'a dictmap = fun s v dict ->
  let rec descend = function
    | [], Node (_, subs) -> Node (Some v, subs)
    | c :: cs, Node (present, []) ->
      Node (present, [ (c, descend (cs, empty)) ])
    | c :: cs as s, Node (present, (subc, subdict) :: rest) ->
      if c = subc then
        Node (present, (subc, descend (cs, subdict)) :: rest)
      else
        let Node (_, nrest) =  descend (s, Node (present, rest)) in
        Node (present, (subc, subdict) :: nrest)
  in
  descend (explode s, dict)
```

Solution to question 2.5

```
let rec main dict =
  match Str.(split (regexp "[␣\t]+") (read_line ())) with
  | [ "get" ; word ] ->
    begin try
        print_endline (get word dict)
      with Not_found ->
        print_endline "Not_found."
    end ;
    main dict
  | [ "set" ; word ; value ]  ->
    let dict = set word value dict in
    print_endline "Added." ;
    main dict
  | [ "exit" | "quit" | "bye" ] ->
    print_endline "Bye."
  | _ ->
    print_endline "Unknown␣command." ;
    main dict

let  () = main empty
```

Solution to exercise 3

Nothing much to change to the code actually, except for lifting the calls to explode and updating the type definition and annotations.

```ocaml
type ('a, 'b) trie = Node of 'b option * ('a * ('a, 'b) trie) list

let empty = Node (None, [])

let get = fun s dict ->
  let rec descend = function
    | [], Node (Some value, _) -> value
    | [], Node (None, _) -> raise Not_found
    | _ :: _, Node (_, []) -> raise Not_found
    | c :: cs as s, Node (present, (subc, subdict) :: rest) ->
      if c = subc then
        descend (cs, subdict)
      else
        descend (s, Node (present, rest)) in
  descend (s, dict)

let set = fun s v dict ->
  let rec descend = function
    | [], Node (_, subs) -> Node (Some v, subs)
    | c :: cs, Node (present, []) ->
      Node (present, [ (c, descend (cs, empty)) ])
    | c :: cs as s, Node (present, (subc, subdict) :: rest) ->
      if c = subc then
        Node (present, (subc, descend (cs, subdict)) :: rest)
      else
        let Node (_, nrest) =  descend (s, Node (present, rest)) in
        Node (present, (subc, subdict) :: nrest)
  in
  descend (s, dict)

let explode s =
  let rec loop acc i =
    if i < 0 then acc else loop (String.get s i :: acc) (pred i) in
  loop [] (String.length s - 1)

let rec main dict =
  match Str.(split (regexp "[␣\t]+") (read_line ())) with
  | [ "get" ; word ] ->
    begin try
        print_endline (get (explode word) dict)
      with Not_found ->
        print_endline "Not_found."
    end ;
    main dict
  | [ "set" ; word ; value ]  ->
    let dict = set (explode word) value dict in
    print_endline "Added." ;
    main dict
  | [ "exit" | "quit" | "bye" ] ->
    print_endline "Bye."
  | _ ->
    print_endline "Unknown␣command." ;
    main dict
```

```

let () = main empty
```