22c:111 (CS:3820) Programming Language Concepts
Fall 2014
Homework Assignment 2
Due: Monday, Oct 20 at 11.59pm

Predictive Text with T9

T9 stands for *Text on 9 Keys* and is a predictive text technology for mobile phones — or it was back in the days when dinosaurs roamed the earth and mobile phones only had a handful of actual keys.

Each of the keys 2 to 9 is associated with three or four characters, see Fig. 1a for the most common mapping. The user presses a sequence of keys and, by referring to a dictionary, an algorithm tries to predict which word composed of the possible combinations of characters associated to the sequence of keys the user meant to type. While the dictionary of words comes populated with some common words, a user can add new words to it, and the algorithm can learn the most frequently chosen words to improve the accuracy of its predictions. For this exercise we will program a predictive text algorithm for T9 in OCaml.

About this assignment To map sequences of keys to words in the dictionary we will use a data structure called *trie*, which is also known as *prefix tree* or *radix tree*, see Fig. 1b for an example. A trie is a tree, where each node has zero or more child nodes. The edges between nodes are labeled with exactly one symbol. Each node contains a list of words that can be produced by the sequence of symbols from the root of the tree.

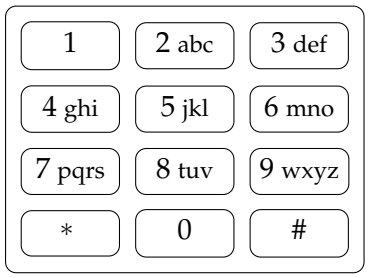
We will proceed in small steps towards a final solution, starting with basic functions and composing them towards the final algorithm. Later questions depend on your solutions to earlier questions, and earlier questions are easier. Therefore you must solve the questions in order. Do not use functions from the standard library of OCaml or any other library. Your code must be executable on its own. Make sure that your functions are type correct and have the signature given.

Test your functions to make sure that they work exactly as specified in the following questions. We will automatically grade your submission based on its compliance with our hidden test cases.

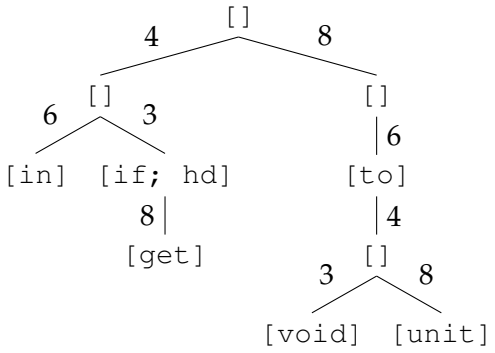
Submission instructions Type your code into the file `hw2.ml` and submit this file with exactly this name in the dropbox on ICON. Do not delete any functions in that file and make sure that it compiles. Otherwise our automatic grading scripts will fail and you may get no points at all.

You will find two additional files with this assignment, `checker.ml` and `Makefile`. Do not modify or submit these, they are meant to give you a preview on how your submission is assessed. We will use different test cases for the actual grading. Open a terminal window in Linux or Mac OS X, or use the Cygwin terminal in Windows, and type `make test` to run some test cases on the file `hw2.ml` in the same directory. Of course, you can also work in the OCaml toplevel to test your functions, but make sure to copy them to `hw2.ml` and run `make test` at least once before you submit.

You may want to use an editor with OCaml support. I recommend Emacs with the Tuareg mode. Alternatives are Vi and Eclipse with the ODT plugin, but I do not have experience with them.



(a) Numeric keys and associated characters



(b) Trie containing a few short words

Figure 1: A keypad and trie of a dictionary for T9

You will be perfectly fine with any other editor, however, features such as syntax highlighting will probably make you more productive, in particular as a first-time user of OCaml.

1 Words to Key Sequences (8 points)

Given a word we want to compute the sequence of keys producing that word. Let us write two functions for this purpose.

1. The first function `int_of_char` takes a value of type `char` and returns a value of type `int` as defined by the common mapping of characters to keys, see Fig. 1a. You may assume that all characters are in lower case. If a character outside the range `'a'` to `'z'` is given as an argument, return the expression `raise Not_found`, which raises the built-in exception `Not_found`.

   (4 points)

```
let int_of_char char = (* ... *)
val int_of_char : char -> int = <fun>
```

2. Then we need a function that takes a word as input and returns a value of type `int list`, which is the sequence of keys producing the word. For simplicity we consider strings of type `char list` instead of the more natural and built-in choice `string`. Use the function `int_of_char` of the previous question.

   (4 points)

```
let rec intlist_of_string string = (* ... *)
val intlist_of_string : char list -> int list = <fun>
```

2 Association Lists (16 points)

We need an auxiliary data structure to map a key to a value. For simplicity we use an association list: a list of pairs of keys of type `'a` and values of type `'b`. Thus, an association list is of

type `('a * 'b) list`. Given an association list `l` we can construct the association list with the mapping of the key `k` to the value `v` added simply by pushing the pair `(k, v)` to the head of the list as in the expression `(k, v) :: l`.

1. Write a function `assoc` that looks up the binding of a key to a value. Given an argument `key` of type `'a` and an association list `dict` of type `('a * 'b) list` return the value `v` of type `'b` from the first pair `(k, v)` in the list, where `k` is equal to `key`. If there is no such pair, return the expression `raise Not_found`, which raises the pre-defined exception `Not_found`. (6 points)

   ```
   let rec assoc key dict = (* ... *)
   val assoc : 'a -> ('a * 'b) list -> 'b = <fun>
   ```

2. Is your function tail recursive? Why, or why not? How could you write a tail recursive version? (2 points)

3. Write a function `change` that returns an association list with the binding of a key to a new value. Given the arguments `key` of type `'a`, `value` of type `'b` and an association list `dict` of type `('a * 'b) list` return an association list of the same type where the first binding of `key` to some value is replaced with a binding of `key` to `value`. If the association list does not contain a binding to `key`, return the association list `dict` with the mapping of `key` to `value` added. (8 points)

   ```
   let rec change key value dict = (* ... *)
   val change : 'a -> 'b -> ('a * 'b) list -> ('a * 'b) list = <fun>
   ```

3 Tries (30 points)

We represent a trie with the following data type

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

A trie is parametrized by the type `'a` of its edge labels and the type `'b` of the elements stored at a node. It has a single constructor `Node`, which is necessary to make the recursive type well-founded. We can return the list of elements of the root node with the following function.

```
let words trie =
  match trie with
    | Node (words, _) -> words

val words : ('a, 'b) trie -> 'b list = <fun>
```

The second parameter of the constructor `Node` is an association list of edge labels to the tries they lead to. We can access this association list with the following function.

```
let branches trie =
  match trie with
    | Node (_, branches) -> branches

val branches : ('a, 'b) trie -> ('a * ('a, 'b) trie) list = <fun>
```

1. Give a value for the empty trie without edges out of the root and with no data stored in its root node. (2 points)

```
let empty = (* ... *)
```

2. Write a function that is given a trie and an edge label as parameters and returns the trie at the end of the edge. Use the function `assoc` from above. Since this function will raise the runtime error `Not_found` when there is no edge with the given label, we need to catch this error and return an alternative value in this case.

   Error handling in OCaml is implemented with **`try`** and **`with`** keywords. The code fragment

```
try 1 / x with Division_by_zero -> 0
```

   evaluates to `1/x` unless the exception `Division_by_zero` is caught. In this case it evaluates to `0`.

   Use this structure to catch the runtime error `Not_found` that may be raised by `assoc` and return the empty trie in this case. (6 points)

```
let trie_of_key trie key = (* ... *)
val trie_of_key : ('a, 'b) trie -> 'a -> ('a, 'b) trie = <fun>
```

3. Write a function `find` that is given a trie and a sequence of keys and returns the list of words in the trie for that key sequence. Traverse the trie from the root, following edges to sub-tries and return the list of words of the node you arrive at when the whole sequence of keys has been consumed. Use the function `trie_of_key` from the previous question. (8 points)

```
let rec find trie keys = (* ... *)
val find : ('a, 'b) trie -> 'a list -> 'b list = <fun>
```

4. Write two auxiliary functions: `add_word` takes a word and a trie and adds the word at the head of the list of words at the root of the trie, and `replace` takes a key and two tries, `branch` and `trie` and replaces the mapping of the key to its sub-trie in `trie` by `branch`. Inside the `replace` function use the function `change` from a previous question. (6 points)

```
let add_word word trie = (* ... *)
val add_word : 'a -> ('b, 'a) trie -> ('b, 'a) trie = <fun>

let replace key branch trie = (* ... *)
val replace :
  'a -> ('a, 'b) trie -> ('a, 'b) trie -> ('a, 'b) trie = <fun>
```

5. Finally write a function `add` to insert a word into the dictionary of a trie. Use the functions `add_word` and `replace` from the previous question.

   (8 points)

```
let rec add keys word trie = (* ... *)
val add : 'a list -> 'b -> ('a, 'b) trie -> ('a, 'b) trie = <fun>
```