Datatypes and Trees

1 Exceptions

In OCaml, *run-time errors* are reported with *exceptions*. An exception can be defined for later use, with the syntax: **exception** *name*. The exception can also accept arguments of some specified type, using the syntax: **exception** *name* **of** *type*. Such an exception is called with (*name argument*); for example, the exception `(Invalid_argument "string")` must take a single string as its argument. Raise an exception with the keyword **raise**. See examples in the functions "take" and "drop", defined in the lists chapter.

It is also possible to *handle* an exception with an *exception handler*. Note that the types must be consistent with the type of the function.

```
safe_divide : int -> int -> int

let safe_divide x y =
  try x / y with
    Division_by_zero -> 0
```

2 Datatypes

It is possible to declare your own datatypes with the syntax `type name = constructor1 of type1 | constructor2 of type2 | ...`, where *constructors* are the possible forms that the type can take. The constructors must start with a capital letter. It is possible to pattern match on these, as with built-in types:

```
type colour =
   Red
 | Green
 | Blue
 | RGB of int * int * int

components : colour -> int * int * int

let components c =
  match c with
    Red -> (255, 0, 0)
  | Green -> (0, 255, 0)
  | Blue -> (0, 0, 255)
  | RGB (r, g, b) -> (r, g, b)
```

This is analagous to pattern matching some expression to different integers.

A type can be polymorphic; that is, a part of a type (called a *type variable*) can vary. For example, `type 'a option = None | Some of 'a`. In words, a value of type α `option` is either nothing or something of type α. This can be especially useful in exception handling, as we shall see below.

A type can also be defined recursively. For example, `type 'a sequence = Nil | Cons of 'a * 'a sequence`. Note that this has a direct mapping onto the built-in list type, where `[]` is `Nil`, `[1]` is `Cons (1, Nil)` and `['a', 'b']` is `Cons ('a', Cons ('b', Nil))` etc. All the functions we defined using lists can easily be converted to ones using our newly defined sequence.

Another example of a recursive type is set up below:

```
type expr =
  Num of int
| Add of expr * expr
| Subtract of expr * expr
| Multiply of expr * expr
| Divide of expr * expr
| Power of expr * expr

evaluate : expr -> int
evaluate_opt : expr -> int option

let rec evaluate e =
match e with
  Num x -> x
| Add (e, e') -> evaluate e + evaluate e'
| Subtract (e, e') -> evaluate e - evaluate e'
| Multiply (e, e') -> evaluate e * evaluate e'
| Divide (e, e') -> evaluate e / evaluate e'
| Power (e, e') -> power (evaluate e) (evaluate e')

let evaluate_opt e =
  try Some (evaluate e) with Division_by_zero -> None
```

Note that we have used our `option` type for error handling. Thus, 1 + 2 * 3 would be represented as `Add (Num 1, Multiply (Num 2, Num 3))` in the expr type.

3 Trees

A *binary tree* is used to represent structures that branch. Let us define the *data structure* as follows:

```
type 'a tree =
  Br of 'a * 'a tree * 'a tree
| Lf
```

This is a polymorphic data structure where the branches hold a value and the left and right subtrees. A leaf occurs when there is no left and no right subtree. Thus, a valid tree would be `Br (2, Br (1, Lf, Lf), Lf)`.

We shall now define various functions for trees.

```
size : 'a tree -> int
total : int tree -> int

let rec size tr =
  match tr with
    Br (_, l, r) -> 1 + size l + size r
  | Lf -> 0

let rec total tr =
  match tr with
    Br (x, l, r) -> x + total l + total raise
  | Lr -> 0
```

The depth of a tree is the longest path from the root of the tree to a leaf.

```
max : int -> int -> int
maxdepth : 'a tree -> int
list_of_tree : 'a tree -> 'a list
tree_map : ('a -> 'b) -> 'a tree -> 'b tree

let max x y =
  if x > y then x else y

let rec max_depth tr =
  match tr with
    Br (_, l, r) -> 1 + max (maxdepth l) (maxdepth r)
  | Lf -> 0

let rec list_of_tree tr =
  match tr with
    Br (x, l, r) -> list_of_tree l @ [x] @ list_of_tree r
  | Lf -> []

let rec tree_map f tr =
  match tr with
    Br (x, l, r) -> Br (f x, tree_map f l, tree_map f r)
  | Lf -> Lf
```

We shall now discuss a more specific kind of tree called a *binary search tree*. This is a tree such that all branches with values smaller than any given branch are to the left of said branch and all branches with values larger than a given branch are to the right of said branch. It can be used to implement a dictionary with more efficient lookup times (O(log n) instead of O(n)).

```
lookup : ('a * 'b) tree -> 'a -> 'b option
insert : ('a * 'b) tree -> 'a -> 'b -> ('a * 'b) tree
tree_of_list : 'a list -> 'a tree

let rec lookup tr k =
  match tr with
    Lf -> None
  | Br ((k', v'), l, r) ->
      if k = k' then Some v
      else if k < k' then lookup l k
      else lookup r k

let rec insert tr k v =
  match tr with
    Lf -> Br ((k, v), Lf, Lf)
  | Br ((k', v'), l, r) ->
      if k = k' then Br ((k, v), l, r)
      else if k < k' then Br ((k', v'), insert l k v, r)
      else Br ((k', v'), l, insert r k v)

let rec tree_of_list l =
  match l with
```

```
    [] -> Lf
  | (k, v)::t -> insert (tree_of_list t) k v
```

Note that `tree_of_list` can be implemented more efficiently using tail recursion.

Strings are sequences of characters enclosed by double quotes, with the built-in type **string**.