(*

Using Lists as a representation for data structures *)

(* Searching for an element `a` in a list `s`

Linear search: When should we use it?
When we cannot use any property of the *type* of elements

TIME COMPLEXITY: *O(n)* where *n* = `length s` in the worst case
Question: What is the "average case" time complexity? What does "average case" mean?
*)

```
exception NotFound;;
let rec find a s = match s with
        [ ] -> raise NotFound
    | x::xs -> if a = x  then true
                         else find a xs
;;
```

(* **Examples** *)

```
find 3 [];;
find 3 [1;2;3;4;5];;
find 6 [1;2;3;4;5];;
```

(* **Exercise**: Modify the above program `find` to return the first index, counting from 0, at which `a` appears in `s` and raising an exception if it does not appear. What is the time complexity?
*)

(* **Exercise**: Modify the above program `find` to return the sequence of all indices, ranging from 0, at which `a` appears in `s`. What should you do if `a` does not appear in `s`?
*)

(* **Exercise**: Modify the above program `find` to write a program `member a s` which returns `true` if `a` appears in `s` and `false` otherwise.
What is the time complexity?
*)

(*
Numerals of unbounded size *)

(* Binary representation of numerals
— better than unary since representation takes *O(lg n)* space, instead of *O(n)*
— some operations are significantly more efficient: we will code and analyse these below
*)

(* We will use integers `0` and `1` (of type `int`) for bits — instead of `bool` values;
this is for convenience, so that one or two functions can be easily written.
The downside is that the OCaml interpreter may complain that our case analysis is not exhaustive
*)

(* **Representational Invariants**

Representing a numeral as a bit sequence
Design Choice: Least Significant Bit to Most Significant Bit order: "Little-endian"

REPRESENTATIONAL INVARIANT:
For s = [x_0 ; … ; x_n]
require that *for all* $0 \le i <$ (length s), ($x_i = 0$ or $x_i = 1$)

Note: n = (length s)-1
*)

```
type big = int list;;

let rec validbig s = match s with
    [ ] -> true
| x::xs ->  (x=0 || x=1) && (validbig xs)
;;
```

(* **Examples** *)

```
validbig [];;
validbig [0];;
validbig [1];;
validbig [1; 0];;
validbig [1; 1];;
validbig [0; 0; 0];;
validbig [0; 1; 0];;
validbig [0; 0; 1];;
validbig [1; 0; 0];;
validbig [1; 1; 0];;
```

```
validbig [1; 0; 1];;
validbig [1; 1; 1];;
```

(* **Exercise**: Write the function `validbig: big -> bool` using `map` and `fold_left`
*)

(* **Standard meaning** of `s: big = [` x_o`;` ...`;` x_n `]` is given by function
`big2int : big -> int` defined as

$$\texttt{big2int s} = \Sigma_{i=0}^{n} x_i * 2^i$$

*)

```
let rec big2int s = match s with
    [ ] -> 0
  |  x::xs -> x + 2*(big2int xs)
;;
```

(* **Examples** *)

```
big2int [];;
big2int [0];;
big2int [1];;
big2int [1; 0];;
big2int [1; 1];;
big2int [0; 0; 0];;
big2int [0; 1; 0];;
big2int [0; 0; 1];;
big2int [1; 0; 0];;
big2int [1; 1; 0];;
big2int [1; 0; 1];;
big2int [1; 1; 1];;
```

(* **Multiple representations for a number!**
Note that this representation allows for multiple representations of the same number e.g., zero can be represented as `[ ]` or `[0]` or `[0;0]` ...

So we strengthen the representational invariant to disallow redundant trailing `0` bits (note these are MSBs), i.e., we conjoin the condition: $x_n \neq 0$
*)

(* **Exercise**: Modify the definition of `validbig` to incorporate the above extra condition.
*)

(* **Addition:** Add two bits with a carry bit,
returns a pair of bits — a "carry bit" and a "place bit" *)
(* *Assumption*: each of `x, y, c` are either `0` or `1` *)

```
let addc x y c =
            let sum = x+y+c
```

```
                           in   (sum / 2, sum mod 2)
    ;;
```

(* Examples *)

```
    addc 0 0 0;;
    addc 0 0 1;;
    addc 0 1 0;;
    addc 1 0 0;;
    addc 1 0 1;;
    addc 1 1 0;;
    addc 0 1 1;;
    addc 1 1 1;;
```

(* **Correctness** condition for `addc`:
    for all `x,y,c: bit,` if `(c',b) = addc x y c` then `x+y+c = 2*c' + b`
*)

(* *Carry propagation*: add a carry bit `c` to a `big` value represented by the list `s,` and propagate the carry if necessary. *)

```
    let rec propcarry s c = if c = 0
                            then s (* no propagation if c = 0 *)
                            else match s with  (* note c = 1 *)
                              [ ] -> [c]     (* propagate the 1 *)
                            | x::xs -> if x+c=1 (* c=1 so x=0 *)
                                       then 1::xs
                                                 (* make LSB 1 *)
                                        else 0::(propcarry xs 1)
               (* c=1, x=1, so LSB is 0, recur with carry = 1 *)
    ;;
```

(* **Examples** *)

```
    propcarry [ ] 0;;
    propcarry [ ] 1;;
    propcarry [0] 1;;
    propcarry [1] 1;;
    propcarry [1; 0; 1] 0;;
    propcarry [1; 0; 1] 1;;
    propcarry [ 1; 1; 1] 1;;
```

(* **Exercise**: State and prove the correctness of `propcarry s c` *)

(* **Addition**: add two `big` values — in LSB to MSB — with an initial carry bit `c` *)

```
    let rec addbig s1 s2 c = match s1 with
                              [ ] -> propcarry s2 c
                            | x::xs -> (match s2 with
                                         [ ] -> propcarry s1 c
                                       | y::ys ->
                                          let (c', b) = addc x y c
                                           in b::(addbig xs ys c'))
    ;;
```

(* **Examples** *)

```
addbig [ ] [ ] 1;;
addbig [ ] [ ] 0;;
addbig [ ] [1; 1; 1] 1;;
addbig [1; 1; 1] [1; 1; 1 ] 1;;
addbig [1; 1; 1] [1; 1; 1 ] 0;;
addbig [1; 1; 1 ] [] 1;;
```

(* **Check** that given two valid `big` values and a carry bit, the result of `addbig` is a valid representation of a value in `big`. That is, verify that the Representational Invariant is maintained. *)

(* How will you prove this? *)

(*

Corrrectness of `addbig`

for all `c: bit`, *for all* `s1: big`, *for all* `s2: big`,
`big2int (addbig s1 s2 c) = (big2int s1) + (big2int s2) + c`

Proof: by <u>induction</u> on `max (length s1) (length s2)`

Base case: `max (length s1) (length s2) = 0`
Analysis implies that `s1 = [ ]` and `s2 = [ ]`

```
  big2int ( addbig [ ] [ ] c )
= big2int ( propcarry [ ] c )
```
// <u>subcase</u> `c = 0`
```
=  big2int ( [ ] )
=  0
```
// <u>subcase</u> `c = 1`
```
=  big2int ( [ 1 ] )
=  1
```

Induction Hypothesis: Assume that *for all* `s1': big, s2': big, c: bit,` such that `max (length s1') (length s2')` $= k$
`big2int (addbig s1' s2' c) = (big2int s1')+(big2int s2')+c`

Induction Step: Let `max (length s1) (length s2)` $= k+1$
// <u>Case</u> `(length s1)` $= k+1$ and `(length s2)` $\leq k+1$
Analysis implies that `s1` is of the form `x::xs`

```
  big2int ( addbig x::xs s2 c )
```
// (<u>subcase</u> `s2 = [ ]`)
```
=  big2int ( propcarry s1 c )
=  (big2int s1) + c
```
// correctness of `propcarry`
```
= (big2int s1) +  0 + c
= (big2int s1) + (big2int s2) +  c // big2int
```

```
// (subcase s2 = y::ys)  // length ys ≤ k
 =   big2int (b::(addbig xs ys c' ) where (c', b) = addc x y c
 =   b + 2 * ( addbig xs ys c' )  // big2int
 =  b + 2 * ((big2int xs) + (big2int ys) +  c') // IH
 = ( b + 2*c' ) + 2*((big2int xs) + (big2int ys)) // +distr
 = (x + y + c) + 2*((big2int xs) + (big2int ys)) //addc
 =  x+ 2*(big2int xs) + y + 2*(big2int ys) + c // rearranging
 = (big2int (x::xs)) + (big2int (y::ys)) + c // big2int

//   Case (length s2) = k+1 and (length s1 ≤ k — left as an exercise
*)

(* Properties to prove about addbig *)

(* Commutativity
 for all s1, s2: big,
      addbig s1 s2 0 = addbig s2 s1 0
*)

(* Associativity
 for all s1, s2, s3: big,
 addbig (addbig s1 s2 0) s3 0 = addbig s1 (addbig s2 s3 0) 0
*)

(* Identity
 for all s: big,
   addbig s [ ] 0 =  s = addbig [ ] s 0
*)



(* Multiply two big values in LSB to MSB representation *)
     let rec multbig s1 s2  = match s1 with
            [ ] ->  [ ]
            |  x::xs -> (match s2 with
                          [ ] ->  [ ]
                        | y::ys -> let zs = multbig xs s2
                                    in
                                      (match x with
                                       0 -> 0::zs
                                      | 1 ->  addbig s2 (0::zs) 0
                                       )
                         )
     ;;


(* Examples *)
     multbig [ ] [1; 1; 1];;
     multbig [1; 1; 1 ] [];;
```

```
multbig [ 1 ] [1; 1; 1];;
multbig [1; 1; 1] [1 ];;
multbig [1; 1; 1] [1; 1; 1 ];;
multbig [1; 0; 1] [1; 1; 1 ];;
multbig [1; 0; 1] [1; 0; 1 ];;
```

(* **Corrrectness** of `multbig`

for all `s1: big`, *for all* `s2: big`,
`big2int (multbig s1 s2) = (big2int s1) * (big2int s2)`

Proof by structural induction on `s1`
Base Case `(s1 = [ ])`

```
   big2int (multbig [ ] s2)
 = big2int [ ]
 = 0
 = 0 * (big2int s2)
 = (big2int [ ]) * (big2int s2)
```

Induction Hypothesis: Assume that
for all `s1': big` such that `(length s1)` = k, *for all* `s2'`
`big2int (multbig s1' s2') = (big2int s1') * (big2int s2')`

Induction Step
// Case `(length s1)` = k+1
Analysis `s1` is of the form `x::xs`
// Subcase analysis on `s2`
// subcase `s2 = [ ]`

```
   big2int (multbig s1 [ ])
 = big2int  [ ]   // multbig
 = 0              //big2int
 = (big2int s1) * 0  // annihilator of *
 = (big2int s1) * (big2int s2)
```

// subcase `s2` is of the form `y::ys`

```
  big2int (multbig (x::xs) (y::ys) )
```

// subsubcase `x=0`

```
 = big2int (0::(multbig xs s2))
 = 2* (big2int (multbig xs s2)) // big2int
 = 2* ((big2int xs) * (big2int s2))// IH
 = (2*(big2int xs)) * (big2int s2)// associativity of *
 = (big2int (0::xs)) * (big2int s2) // big2int
 = (big2int (x::xs)) * (big2int s2) // x=0
 = (big2int s1) * (big2int s2). // s1 = x::xs
```

// subsubcase `x=1`

```
 = big2int ( addbig s2 (0::(multbig xs s2)) 0)
 = (big2int s2) + 2* (big2int (multbig xs s2)) + 0
```

```
        = (big2int s2) + 2* ((big2int xs) * (big2int s2))// IH
        = (1+2*(big2int xs)) * (big2int s2)// rearranging
        = (big2int (1::xs)) * (big2int s2) // big2int
        = (big2int (x::xs)) * (big2int s2)// x=1
        = (big2int s1) * (big2int s2). // s1 = x::xs
*)
```

(* **Properties** to prove about `multbig` *)

(* **Commutativity**
for all `s1, s2: big,`
```
    multbig s1 s2  = multbig s2 s1
*)
```

(* **Associativity**
for all `s1, s2, s3: big,`
```
    multbig  (multbig s1 s2) s3 = multbig s1 (multbig s2 s3)
*)
```

(* **Identity**
for all `s: big,`
```
    multbig s [1] = s =  multbig [1] s
*)
```

(* **Annihilator** of multiplication
for all `s: big,`
```
    multbig s [] = [] = multbig [] s
*)
```

(* **Distributivity** of multiplication over addition
for all `s1, s2, s3: big,`
```
  multbig (addbig s1 s2) s3 =
                   addbig (multbig s1 s3)(multbig s2 s3)
```

for all `s1, s2, s3: big,`
```
  multbig  s1 (addbig s2 s3) =
       addbig (multbig s1 s2) (multbig s1 s3)
*)
```

(* **Exercise**: Define a function `subbig` that subtracts one `big` value from another.
What might you want to do if the second is larger than the first?
*)

(* **Exercise**: State the semantic correctness condition for function `subbig` *)

(* **Exercise**: Define a function `divbig` that divides one `big` value by another.

Exercise: Define a function `modbig` that returns the remainder when one `big` value is divided by another .
Hint: it might be easier to define both together
*)

(* Exercise: State the semantic correctness condition for functions `divbig` and `modbig` *)

(* **Exercise** (Preservation of Representational Invariant): For all the functions that you define, ensure that if the inputs are valid `big` values then the outputs are valid `big` values *)