(* **Tree data types: Expressions, Abstract syntax trees, Evaluators** *)

(*
We now explore tree data types — by presenting an abstract version of expressions.
*)

```
open List;;
```

(* A data type for a very simple class of expressions can be defined using constructors and recursion *)

```
type exp = N of int
         | Plus of exp * exp
         | Times of exp * exp
;;
```

(*
Note that there are no pointers; no "left child", "right child" in these tree-structured expressions.

The recursively defined type `exp` represents expression trees directly —
it is a recursively defined set with
- denumerably many base cases that can be considered together as one base case, with constructor `N(_)`, parameterised on the value of a numeral, and
- two inductive cases, with constructor `Plus(_,_)` and `Times(_,_)`, each with many combinations as arguments.

For convenience, we use values of type `int` as arguments to the constructor `N( _)`. You may, as an exercise, instead use the `bigint` type.
*)

(* **Examples** *)

```
let e0 = N 7;;
let e1 = Plus(N 3, Times(N 4, N 5));;
let e2 = Times(Plus(N 3, N 4), N 5);;
```

(*

Abstract syntax

Think of `exp` as "operator trees", i.e., the tree representations obtained after parsing and some post-processing. Parenthesization in the syntax is implicit in the tree structure, and is needed only when considering 1-dimensional, i.e., textual representations, rather than 2-d graphical representations.

Observe that we are not interested in the tokens of lexical analysis, so parentheses as explicit syntactic tokens do not concern us in abstract syntax: the parentheses that you see in the OCaml syntactic representations are the markers needed for writing down the 1-dimension representations of preorder traversals of the abstract syntax trees (ASTs). ASTs are what we obtain after we have dealt with parsing — and so we do not need to concern ourselves with precedence issues in the grammar, or conventions such as BoDMAS in deciding which parse tree is intended.

*)

(* **Interlude**: Contrast Abstract Syntax, where the focus is on (tree) structure, with Concrete Syntax, which is defined by a grammar for expressions, e.g.,

```
E —> T          // an expression can be a term
E —> T + E      // an expression can be a term, terminal +, then an expression
T —> F          // a term can be a factor
T —> F * T      // a term can be a factor, terminal *, then a term
F —> n          // a factor can be a numeral;
                //  n stands for a family of terminals, one for each numeral
F —> ( E )      // A factor can be a parenthesised expression;
                //  Note the the left and right parentheses: ( and ) are terminals
```

Concrete syntax comprises production rules of how "surface syntactic" phrases — legitimate sequences of terminal symbols in the language — are produced or recognised by a parser. A parser provides a justification why a phrase is or is not a phrase in a language. Typically, the grammar is a "context-free" grammar — that is the production rules can be applied in any context to generate a sequence of symbols. In the example above, upper case letters E, T and F are non-terminals, whereas all numerals n, the operators + and *, and the parentheses (and) are terminals.

The output of a parser is typically not merely a parse tree (the data structure corresponding to such a justification) but a more processed version, called an abstract syntax tree. Abstract syntax trees typically do not contain parentheses, since the tree captures the essential structure of an syntactic phrase.

*)

(* **Syntactic functions** on tree structured expressions *)

(* `size` returns the number of "nodes" in the tree *)

```
let rec size t = match t with
      N _ -> 1
    | Plus(t1, t2) -> (size t1) + (size t2) + 1
    | Times(t1, t2) -> (size t1) + (size t2) + 1
;;
```

(* Examples *)

```
size e0;;
size e1;;
```

```
size e2;;
```

(* `ht` returns one less than the number of levels in the tree, i.e., the length of the longest path from root to a leaf. *)

```
let rec ht t = match t with
    N _ ->  0
  | Plus(t1, t2) -> (max (ht t1) (ht t2)) + 1
  | Times(t1, t2) -> (max (ht t1) (ht t2)) + 1
;;
```

(* Examples *)

```
ht e0;;
ht e1;;
ht e2;;
```

(* Both `size` and `ht` are good measures for performing induction on trees *)

(* **Standard semantics** — an evaluator, i.e., definitional interpreter, for a simple language of expressions
*)

```
let rec eval t = match t with
    N n ->  n
  | Plus(t1, t2) -> (eval t1) + (eval t2)
  | Times(t1, t2) ->  (eval t1) * (eval t2)
;;
```

(* Examples *)

```
eval e0;;
eval e1;;
eval e2;;
```

(* Note the similarity in the form of the three functions `size`, `ht` and `eval`. This is not incidental. Instead, it reflects a fundamental idea in semantics — whether of programming languages or of logic — namely, that the *meaning of the whole is composed in a well-defined way from the meaning of the parts*. In this structural approach (an early advocate of which was Gottlöb Frege), recursion is used in a very specific way in the functions that analyse the structure of an argument, and the corresponding proofs use structural induction.
*)

(* A prototypical compiler into opcodes for a stack machine *)

```
type opcode = LD of int | PLUS | TIMES;;
```

(* The compiler is similar to a post-order traversal function *)

```
let rec compile t = match t with
  N n ->  [LD (n)]
| Plus(t1, t2) -> (compile t1) @ (compile t2) @ [PLUS]
| Times(t1, t2) ->  (compile t1) @ (compile t2) @ [TIMES]
;;
```

(* Examples *)

```
compile e0;;
compile e1;;
compile e2;;
```

(* **Summary**

We have defined a tiny programming language of expressions, with
- Its abstract syntax as the data type `exp` [Peter Landin 1964-5]
- Its standard evaluator as the recursive function `eval` [Landin 1964-5, Christopher Wadsworth, John Reynolds]
- Some useful "measures" `ht` and `size` on syntax [Landin 1964-5, Gordon Plotkin 1980]
- A prototypical compiler `compile` into opcodes for a stack machine

*)

(*

We now present a stack machine for evaluating postfix traversals of an expression tree. This is a simple abstract machine for programs consisting of sequences of the opcodes defined above.

*)

```
exception Stuck;;
let rec stkmc s code  = match code with
    [ ] ->  s
  |  (LD(n)) :: code' ->  stkmc (n :: s) code'
  |  PLUS :: code' -> (match s with
          n2 :: n1 :: s' ->  let n = n1 + n2
                             in  stkmc (n :: s') code'
     | _ -> raise Stuck)
  |  TIMES :: code' -> (match s with
       n2 :: n1 :: s' ->  let n = n1 * n2
                          in  stkmc (n :: s') code'
     | _ -> raise Stuck)
;;
```

(* The `stkmc` execution of the `compile`-d expression gives us a prototypical abstract machine / virtual machine / calculator.

*)

```
let calculate e = hd (stkmc [] (compile e));;
```

(* Examples *)

```
calculate e0;;
calculate e1;;
calculate e2;;
```

(*

Correctness of the compiler and execution of the stack machine wrt the standard reference semantics given by the definitional interpreter `eval`

for all `e: exp`,

`calculate e = eval e`

*)

(*

Soundness of the "implementation"

for all `e: exp`, *for all* `n: int`,

If `stkmc [] (compile e)` = `[n]` then `eval e = n`

Completeness of the "implementation"

for all `e: exp`, *for all* `n: int`,

If `eval e = n` then `stkmc [] (compile e)= [n]`

*)

(*

Exercise: Extend the integer expression toy language given by the data type `exp`, to include operations such as unary minus, (binary) subtraction, integer division and remainder, absolute value, and any other expressions which you can think of.

Exercise: Next extend the definitional interpreter `eval` to handle all these operations.

Exercise: Then extend the data type `opcode` to handle all new operations, and then extend the function `compile` to generate code for these additional expressions.

Exercise: Extend the definition of the stack machine execution `stkmc` by providing execution rules for the new operations.

Exercise: Ensure that your implementations of these operations are correct with respect to the standard semantics given by `eval`.

*)

(*

A Boolean Evaluator, Calculator, Stack Machine

Exercise: Define a boolean expression language given by the data type `expb`, to include operations such as constants `T` and `F`, (unary) negation `Not`, and binary operations of conjunction `And`, disjunction `Or` (and implication `Imply` and Bi-implication `Iff`, and any other expressions which you can think of.

Exercise: Next extend the definitional interpreter `evalb` to provide a standard semantics to all these boolean expressions.

Exercise: Then define a data type `opcodeb` to encode operations on the booleans, and then define the function `compileb` to generate code for these boolean expressions.

Exercise: Adapt the definition of the stack machine execution to define a function `stkmcb` by providing execution rules for the operations defined above.

Exercise: Ensure that your implementations of these operations are correct with respect to the standard semantics given by `evalb`.
*)