2. A Toy Calculator

Let us revisit the toy calculator, and see how aspects of it may be better expressed in the *functional paradigm*, especially when we have a strong regime of types that are statically determined (but thankfully inferred to a very great extent, sparing us the trouble of having to provide type details in every function).

Having talked earlier of boolean expressions — constants and operations such as conjunction, disjunction and negation — let us expand the toy language. (We will keep growing the language to include new constructs to illustrate new concepts).
The toy language will now have
1. Numeric constants
2. Boolean constants
3. Arithmetic operations
4. Boolean operations
5. Comparison operations on arithmetic expressions, which return boolean results.

(Again, we include only a few typical operators, and leave it to you to grow the language to include other useful arithmetic and comparison operators).

2.0 Abstract Syntax in OCaml

We present the abstract syntax of expressions by defining them as a recursive data type `exp`.

```
type exp = Num of int | Bl of myBool
         |  Plus of exp * exp | Times of exp * exp
         |  And of exp * exp | Or of exp * exp | Not of exp
         |  Eq of exp * exp | Gt of exp * exp
;;
```

1. Numeric constants are represented using the *constructor* `Num`, to which we will provide an OCaml `int` argument (to save us from the tedium of having to encode a syntax for numerals).
2. Boolean constants are represented using the *constructor* `Bl`, to which we will provide a `myBool` argument (to highlight to you that that these abstract expressions are *syntactic* — so we can write `T` and `F`; Also, I am lazy and didn't want to write out `true` and `false`).
3. Arithmetic operations are represented using the *constructors* `Plus` *and* `Times`. Note the capital letters, and that these take (sub)expressions as arguments.
4. Boolean operations are represented using the *constructors* `And`, `Or` *and* `Not` which take (sub)expressions as arguments.
5. Comparison operations using the *constructors* `Eq` *and* `Gt`, which take (sub)expressions as arguments. The operations represent respectively (i) equality of two numeric expressions; (ii) whether the first numeric expression is strictly greater than the second. They are meant to return a boolean value.

Some expressions that we can write in our abstract syntax are given below, as "*terms*" in the data type `exp`. A term is a "1-dimensional" representation of an "*abstract syntax tree*" (AST), and it is instructive to think of terms as trees with teh constructors as the root nodes of (sub)trees.

```
let test1 = Plus (Times (Num 3, Num 4),
                  Times (Num 5, Num 6));;
let test2 = Or (Not (Bl T),
                And (Bl T,
                     Or(Bl F,
                        Bl T)));;
let test3 = Gt (Times (Num 5, Num 6),
               (Times (Num 3, Num 4)));;
let test4 = And (Eq(test1, Num 42), Not test3);;
```

Functions from the data type exp

We first define two useful "measure" functions on the data type `exp` — `ht`, which returns the "height" of a syntax tree (counting leaves as being of height zero), and `size`, which returns the number of nodes in the abstract syntax tree. Both are recursive functions, which follow the structure of the trees (so the typical way to reason about these functions is by induction). Note the similarity between the two functions. In `ht` we map the the base cases to the value 0, whereas in `size`, they are mapped to the value 1. The induction cases in both functions involve recursive calls on the subexpressions (subtrees). In function `ht`, each construction is associated with 1 plus the (maximum of the) recursive call to `ht` on the subtrees, whereas in `size` they are associated with 1 plus the (sum of the) recursive call to `size` on the subtrees

```
let rec ht e = match e with
    Num n  -> 0
  | Bl b -> 0
  | Plus (e1, e2)  -> 1 + (max (ht e1) (ht e2))
  | Times (e1, e2)  -> 1 + (max (ht e1) (ht e2))
  | And (e1, e2)  -> 1 + (max (ht e1) (ht e2))
  | Or (e1, e2)  -> 1 + (max (ht e1) (ht e2))
  | Not e1 -> 1 + (ht e1)
  | Eq (e1, e2)  -> 1 + (max (ht e1) (ht e2))
  | Gt(e1, e2)  -> 1 + (max (ht e1) (ht e2))
;;

let rec size e = match e with
    Num n  -> 1
  | Bl b -> 1
  | Plus (e1, e2)  -> 1 +  (size e1) + (size e2)
  | Times (e1, e2)  -> 1 +  (size e1) + (size e2)
  | And (e1, e2)  -> 1 +  (size e1) + (size e2)
  | Or (e1, e2)  -> 1 +  (size e1) + (size e2)
  | Not e1 -> 1 + (size e1)
  | Eq (e1, e2)  -> 1 +  (size e1) + (size e2)
  | Gt(e1, e2)  -> 1 +  (size e1) + (size e2)
;;
```

Let us test the function `ht` and `size` on the four examples that we defined above.

```
let h1 = ht test1;;
```

```
let h2 = ht test2;;
let h3 = ht test3;;
let h4 = ht test4;;

let s1 = size test1;;
let s2 = size test2;;
let s3 = size test3;;
let s4 = size test4;;
```

Do we get the expected answers?

A definitional interpreter for ASTs in data type `exp`

$eval\llbracket\ \underline{N}\ \rrbracket = n$
$eval\llbracket \mathrm{T} \rrbracket = true$ and $eval\llbracket \mathrm{F} \rrbracket = false$

$eval\llbracket \underline{E_1 + E_2} \rrbracket = eval\llbracket \underline{E_1} \rrbracket + eval\llbracket \underline{E_2} \rrbracket$
$eval\llbracket \underline{E_1 * E_2} \rrbracket = eval\llbracket \underline{E_1} \rrbracket \times eval\llbracket \underline{E_2} \rrbracket$
(where $+, \times$ represent integer addition and multiplication).

$eval\llbracket \underline{E_1 \wedge E_2} \rrbracket = eval\llbracket \underline{E_1} \rrbracket \ \&\&\ eval\llbracket \underline{E_2} \rrbracket$
$eval\llbracket \underline{E_1 \vee E_2} \rrbracket = eval\llbracket \underline{E_1} \rrbracket \ ||\ eval\llbracket \underline{E_2} \rrbracket$
$eval\llbracket \underline{\neg E_1} \rrbracket = not\ (eval\llbracket \underline{E_1} \rrbracket)$
(where $\&\&$, $||$, not represent boolean conjunction, disjunction and negation).

$eval\llbracket \underline{E_1 = E_2} \rrbracket = eval\llbracket \underline{E_1} \rrbracket =^? eval\llbracket \underline{E_2} \rrbracket$
$eval\llbracket \underline{E_1 > E_2} \rrbracket = eval\llbracket \underline{E_1} \rrbracket >^? eval\llbracket \underline{E_2} \rrbracket$
(where $=^?$, $>^?$ represent equality and greater-than comparisons on integers).

Coding *eval* in OCaml

By considering two different kinds of constants, numeric and boolean, we have created a minor complication regarding the set of values. OCaml doesn't allow us to take a union of two types, unless we explicitly tag the values from the two types with constructors to mark which original type they belong to. We use two constructors `N` on `int`s and `B` on `bool`s to define a type called `values`. (As we enrich our language, this type will become more and more complex and interesting).

```
type values = N of int | B of bool ;;

let rec eval e = match e with
    Num n   -> N n
  | Bl b -> B (myBool2bool b)
  | Plus (e1, e2)   ->  let N n1 = (eval e1)
                       and  N n2 = (eval e2)
                         in N (n1 + n2)
  | Times (e1, e2)  ->  let N n1 = (eval e1)
```

```
                        and  N n2 = (eval e2)
                          in N (n1 * n2)
  | And (e1, e2)  ->  let B b1 = (eval e1)
                    and  B b2 = (eval e2)
                      in B (b1 && b2)
  | Or (e1, e2)  -> let B b1 = (eval e1)
                  and  B b2 = (eval e2)
                    in B (b1 || b2)
  | Not e1 -> let B b1 = (eval e1) in B (not b1)
  | Eq (e1, e2)  -> let N n1 = (eval e1)
                  and  N n2 = (eval e2)
                    in B (n1 = n2)
  | Gt(e1, e2)  -> let N n1 = (eval e1)
                 and  N n2 = (eval e2)
                   in B (n1 > n2)
;;
```

The OCaml interpreter should raise many warnings (but no errors). Note that in each of the inductive cases, we have used a **`let`** form — we (recursively) evaluate the subexpression(s) to a *value* which will be marked by either the constructor `N` or `B` depending on whether it is a numeric or boolean value that is being returned. *Pattern-matching* will allow us to match the expected form, extract the (actual) answer(s) for use in returning the (tagged) computed result. The keyword **`and`** indicates that the two subexpression evaluations (and **`let`** bindings) can be done in parallel. The "**`let`** _ **`in`** _" form allows us to make local bindings (so the `n1, n2, b1, b2` are *not* visible outside the individual cases.

However, the OCaml interpreter will report that the assumed patterns which we have used are not exhaustive and that there may be other cases which we have not considered. (Actually we have considered all possible cases for correctly formed expressions, where we are careful not to use numeric expressions where booleans are expected and conversely. The discussion on how to ensure that do not make such errors will be dealt with when we discuss "*type-checking*").

Do take a look at the definition of the function eval, and compare and contrast it with the definitions of `ht` and `size` (apart from the use of the "**`let`** _ **`in`** _" form.) Note the overall schematic similarity. Note also that we use `myBool2bool` to convert the syntactic `T, F` to OCaml `bool` values.

Let us satisfy ourselves that `eval` seems to be correctly defined, by trying it on our examples:

```
let v1 = eval test1;;
let v2 = eval test2;;
let v3 = eval test3;;
let v4 = eval test4;;
(* You may try let v5 = eval (And(test1, test4));;  *)
```

Compiling expressions for a Stack machine

Let us first extend the opcodes for the stack machine. Note that in OCaml, we can define the set of opcodes as a data type, with each opcode expressed as a constructor (we prefer to use all capitals).

```
type opcode = LDN of int | LDB of bool
            | PLUS | TIMES | AND | OR | NOT | EQ | GT;;
```

The compiler is very natural express as a recursive function from the type `exp` to `opcode lists`. It is essentially a post-order traversal of the ASTs (see the outputs for the running examples). Again, notice the schematic similarity of the function `compile` to `ht`, `size` and `eval`.

> [Note: Again, we have chosen the expedient of using OCaml's `int` and `bool` types for elements on the stack though strictly speaking we should have been operating with abstract syntactic forms such as numerals.]

```
let rec compile e = match e with
    Num n  -> [ LDN n ]
  | Bl b ->   [LDB (myBool2bool b) ]    (* Constants *)
  | Plus (e1, e2)  ->  (compile e1) @ (compile e2) @ [PLUS]
  | Times (e1, e2)  ->  (compile e1) @ (compile e2) @ [TIMES]
  | And (e1, e2)  ->  (compile e1) @ (compile e2) @ [AND]
  | Or (e1, e2)  -> (compile e1) @ (compile e2) @ [OR]
  | Not e1 -> (compile e1) @ [NOT]
  | Eq (e1, e2)  -> (compile e1) @ (compile e2) @ [EQ]
  | Gt(e1, e2)  -> (compile e1) @ (compile e2) @ [GT]
;;

let c1 = compile test1;;
let c2 = compile test2;;
let c3 = compile test3;;
let c4 = compile test4;;
```

Encoding the Stack Machine in OCaml

The stack machine is easy to code as a tail-recursive function, which takes a stack (represented as a list of `values`), and an opcode list. The case analysis is on the first opcode in the `opcode list`. Of course, when the machine cannot make a move, it has either completed its work (the good case) and returns the `value` at the top of the stack; or else it is *stuck*. We can flag this bad case using OCaml's *exception* mechanism (and in fact, pass the stuck state as an argument to the exception which we unsurprisingly name Stuck.

```
exception Stuck of (values list * opcode list);;

let rec stkmc s c = match s, c with
    v::_, [ ] -> v  (* no more opcodes, return top *)
  | s, (LDN n)::c' -> stkmc ((N n)::s) c'
  | s, (LDB b)::c' -> stkmc ((B b)::s) c'
  | (N n2)::(N n1)::s', PLUS::c' -> stkmc (N(n1+n2)::s') c'
```

```
  | (N n2)::(N n1)::s', TIMES::c' -> stkmc (N(n1*n2)::s’) c'
  | (B b2)::(B b1)::s', AND::c' -> stkmc (B(b1 && b2)::s') c'
  | (B b2)::(B b1)::s', OR::c' -> stkmc (B(b1 || b2)::s') c'
  | (B b1)::s', NOT::c' -> stkmc (B(not b1)::s') c'
  | (N n2)::(N n1)::s', EQ::c' -> stkmc (B(n1 = n2)::s') c'
  | (N n2)::(N n1)::s', GT::c' -> stkmc (B(n1 > n2)::s') c'
  | _, _ -> raise (Stuck  (s, c))
;;
```

Note that I have been liberal in using parentheses, to ensure that nothing is incorrectly associated (which may crop up immediately as OCaml reporting a type error).

Running the stack machine on the output from compile yields the same result as `eval` does for each of our examples. Note however that these are only examples, and correctness of the compiler/abstract machine requires a formal proof, as we did earlier.

```
let w1 = stkmc [ ] (c1) ;;
let w2 = stkmc [ ] (c2) ;;
let w3 = stkmc [ ] (c3) ;;
let w4 = stkmc [ ] (c4) ;;
```