1 Type-checking the Core language

As usual, we model the abstract syntax for locations, expressions and commands as recursive types in Ocaml. Our abstract syntax is slightly different from the book in that we do not implement a separate syntax class for integer - we just fold it right into the expression class using the N constructor.

1.1 Syntax

These are the Ocaml implementations of the abstract syntax trees.

```ocaml
type loc = Loc of int ;;

type expression =
  N of int |
  Deref of loc |
  Add of expression * expression |
  Neg of expression |
  Eq of expression * expression ;;

type command =
  Assign of loc * expression |
  Seq of command * command |
  If of expression * command * command |
  While of expression * command |
  Skip ;;
```

1.2 Typing rules and their implementation

For each syntactic class, there is at least one typing rule for each constructor, there are essentially two rules encoded in one for the Eq constructor.

**Locations:** There is one constructor for locations. The single typing rule for locations simply says a location is well typed if the index is greater than zero.

\[
\begin{align*}
\text{loc} : \text{intloc} & \quad \text{if } i > 0 \\
\end{align*}
\]

We can implement a function checking this as follows:
let well_typed_intloc (Loc i) = i > 0 ;;

Notice that we have destructured the location in the argument. It allows us to get a handle on the integer argument to the Loc constructor. Since there is only one constructor for locations - destructuring in the argument position will never generate a failure because of an incomplete match.

**Expressions:** We break the type-checker for expressions into two functions, one to test if an expression is a well-typed bool expression and another to check for well-typed int expressions. This is OK to do since in any context, we know what type we expect to see.

Consider the rules for int expressions.

<table>
<thead>
<tr>
<th>$N : \text{int}$</th>
<th>$L : \text{intloc}$</th>
<th>$E_1 : \text{intexp}$</th>
<th>$E_2 : \text{intexp}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N : \text{intexp}$</td>
<td>$&amp; L : \text{intexp}$</td>
<td>$E_1 + E_2 : \text{intexp}$</td>
<td></td>
</tr>
</tbody>
</table>

Here is the corresponding code.

```ocaml
let rec well_typed_int_expression e =
  match e with
  | N i -> true
  | Deref l -> well_typed_intloc l
  | Add (e1, e2) -> well_typed_int_expression e1 
    && well_typed_int_expression e2
```

The first intexp rule checks whether a numeral is well-typed. Ocaml's type checker will only allow a value of type int to be used as an argument to the N constructor so, in the Ocaml implementation we, just return the constant true. To implement the dereferencing rule, we must check that the argument to the Deref constructor is a well-typed intloc. The third rule says to check that an expression of the form $E_1 + E_2$ is a well-typed intexp, recursively check that the two addends are well-typed intexp. The Add case implements this by making two recursive calls and computing their logical conjunction (&&).

The type rules for checking whether an expression is a boolexp are as follows:

\[
\begin{align*}
  E : \text{boolexp} & \quad \frac{}{\neg E : \text{boolexp}} \\
  E_1 : \tauexp & \quad E_2 : \tauexp \quad \frac{E_1 = E_2 : \text{boolexp}}{}
\end{align*}
\]

if $\tau \in \{\text{int, bool}\}$
The Ocaml implementation for the negation operator should be obvious - it requires one recursive call. The rule to check whether an expression of the form $E_1 = E_2$ is a well-typed boolean equality is a bit more challenging. It is essentially two rules, one that checks if both $E_1$ and $E_2$ are boolexs and the other that checks that they are both intexps. To be well-typed, it must be one or the other of these two forms so we compute the logical disjunction (||) of conjunctions of the recursive calls.

Here is the Ocaml code implementing these rules:

```ocaml
let rec well_typed_bool_expression e = 
  match e with
  | Neg e -> well_typed_bool_expression e
  | Eq (e1,e2) ->
    (well_typed_int_expression e1 && well_typed_int_expression e2) ||
    (well_typed_bool_expression e1 && well_typed_bool_expression e2)
;;
```

Problem 1.1. Your assignment is to implement the type checker for commands. To do so you will have to complete the implementation of the Ocaml function `well_typed_command` which has type `command → bool`. 