Functions in OCAML

Functions are an integral part of any programming language, even more so for functional programming languages like OCAML.

The discussion of functions always revolves around two parts: *function definition* and *function application*. We will start with function definition.

A function definition needs to accomplish certain things:

- It needs to provide a name for the function
- It needs to provide a symbol for the function's parameter (we will discuss functions of multiple parameters later)
- It needs to provide the *body* of the function. The body is an expression that will be evaluated in a specific environment when the function is applied

Since we already have ways of giving names to functions, we'll want to piggyback on that concept. Hence the syntax for function definitions is as follows:

```
let f x = e
```

where f is the name we want to use for the function, x is the parameter name, and e is the function's body.

The parameter x will typically appear in the expression e. So the typechecker needs to know what type that parameter will be, in order to ensure it is not misused. We will see later that OCAML has an excellent way of figuring out automatically the types of functions without requiring input from the programmer, but for now we will include a type *ascription* to the parameter, like so:

```
let f (x : int) = e
```

Typechecking function definitions

Let us determine how a function definition should be typechecked. The definition is meant to return a "function value". Therefore we would need first and foremost a new type. This is typically called the *arrow type*, and it looks like $t_1 \to t_2$, where t_1 is the type of the argument and t_2 is the type of the return values.

When encountering a function definition, the typechecker needs to determine which arrow type that function has. The type t_1 is easy to determine as it is being provided in the program (via the x : int segment above for example). The typechecker needs to determine the type of t_2. This is obtained by typechecking the expression e in the current static environment extended by a type binding of x to int.

Function definition evaluation

A function definition needs to evaluate to a function value. But there is a bit more needed. The function may use variables that need to be looked at in the dynamic environment. Even though the evaluation of the body expression e will happen much later, when a function application occurs, the environment used for looking up the variables occuring in e is not the one in place when function application occurs, but instead the one in place where the function was defined. This is called **lexical scoping**. For instance in the following code:

```
let y = 2
let f (x : int) = x + y
let y = 3
```

Any application of f later in the code will use the value of 2 for y, and not the value of 3.

What this means is that a "function value" needs to not only contain information about what the parameter x and the expression e are, but it needs to also keep around the dynamic environment at the time when the function was created. This pair of the function together with the environment is called a **closure**.

It is important to emphasize that these closures are regular "values", and you can use them anywhere where you could use other values like integers or booleans. This point has far-reaching consequences, and will be discussed more extensively later.

For now let us close by spelling out how function definition is evaluated: When a function definition is encountered, a new closure object is created, which contains the name of the parameter, the expression that is the body of the function, and the current dynamic environment.

Function application

Along with function definition, we need a way to apply functions. Function application is done by simply placing the value next to the function:

```
f e
```

Here f is any expression that (hopefully) evaluates to a closure, and e is an expression that will evaluate to a value. That value will then be fed in as the value for the argument.

Syntax $e_1 e_2$ where e_1 is an expression that represents the function to be called and e_2 is an expression that represents the value to be applied. In most languages this would have been written as $e_1(e_2)$.

Typechecking We first typecheck e_1, and ensure that it has an arrow type $t_1 \to t_2$. We then typecheck e_2, and we ensure that this has type t_1, as it is meant to represent the input to our function. The whole expression then has type t_2.

Evaluation To evaluate a function application we first evaluate the expression e_1 and ensure we obtain a closure. We then evaluate e_2 and obtain a value v. Now we evaluate the body of the function stored in the closure in the environment that was stored in the closure extended with a binding that binds the parameter x to the value v. The resulting value is the value of the overall expression.

Tuples

Tuples are another useful language construct that allows us to group values/expressions together. These can be of different types, but there is a specific number of them.

Syntax $(e_1, e_2, \ldots, e_n)$ where $e_1, e_2, \ldots$ are arbitrary expressions.

Typechecking We have a new kind of type, called **product type**. If e_1 has type t_1, e_2 has type t_2 and so on, then the type of the overall expression is t1 $*$ t2 $*$ ... $*$ tn.

Evaluation We evaluate each expression in turn, and obtain values v_1, v_2 and so on. The resulting value is then the tuple $(v_1, v_2, \ldots, v_n)$. So notice that we are introducing a new kind of values here, namely tuples of values.

A tuple with only two entries is called a *pair*, one with three entries a *triple* and so on. Tuples with one entry are redundant. Note however that there is a tuple with zero entries, denoted by $()$ and called the unit value. It has the unit type and it is the only value of that type.

"Tuples" can also be used as the parameters to a function. We will make that more precise later on, but for now it allows us to have functions of multiple arguments like so:

```
let f (x, y) = x + y
f (1, 3)
```

The version with a type ascription for the argument looks a bit more awkward:

```
let f ((x, y) : int * int) = x + y
```

Example functions

Here are some practice problems for working with functions:

1. Write a function "lnot" that takes as input a boolean and returns its negation (and use only if-then-else constructs)
2. Write a function "isPositive" that takes as input an integer and returns true if the number is positive
3. Write a function "sign" that takes as input an integer and returns 1 if the integer is positive, 0 if the integer is 0 and -1 if the integer is negative.
4. Write function "order" that takes as input a pair of numbers, and returns a pair of the same two numbers but in increasing order (smallest first)

5. Write a function “areOrdered” that takes as input a triple of integers and returns true if they are in increasing order
6. Write a function “validDate” that takes as input a triple of integers and decides if it is a valid date in format (month, day, year) (counting every year as a non-leap year)