Chapter 3

Case by Case

In the previous chapter, we used the conditional expression **`if`** ... **`then`** ... **`else`** to define functions whose results depend on their arguments. For some of them we had to nest the conditional expressions one inside another. Programs like this are not terribly easy to read, and expand quickly in size and complexity as the number of cases increases.

OCaml has a nicer way of expressing choices – *pattern matching*. For example, recall our factorial function:

```
factorial : int → int

let rec factorial a =
  if a = 1 then 1 else a * factorial (a - 1)
```

We can rewrite this using pattern matching:

```
factorial : int → int

let rec factorial a =
  match a with
    1 -> 1
  | _ -> a * factorial (a - 1)
```

We can read this as "See if `a` matches the pattern `1`. If it does, just return `1`. If not, see if it matches the pattern `_`. If it does, the result is `a * factorial (a - 1)`." The pattern `_` is special – it matches anything. Remember our `isvowel` function from the previous chapter?

```
isvowel : char → bool

let isvowel c =
  c = 'a' || c = 'e' || c = 'i' || c = 'o' || c = 'u'
```

Here is how to write it using pattern matching:

```
isvowel : char → bool

let isvowel c =
  match c with
    'a' -> true
  | 'e' -> true
  | 'i' -> true
  | 'o' -> true
  | 'u' -> true
  | _ -> false
```

If we miss out one or more cases, OCaml will warn us:

```
        OCaml

# let isvowel c =
    match c with
      'a' -> true
    | 'e' -> true
    | 'i' -> true
    | 'o' -> true
    | 'u' -> true;;
# Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
'b'
val isvowel : char -> bool
```

OCaml does not reject the program, because there may be legitimate reasons to miss cases out, but for now we will make sure all our pattern matches are complete. Notice that we had to repeat `true` five times. This would be awkward if the expression to be calculated was more complicated. We can combine patterns like this:

```
isvowel : char → bool

let isvowel c =
  match c with
    'a' | 'e' | 'i' | 'o' | 'u' -> true
  | _ -> false
```

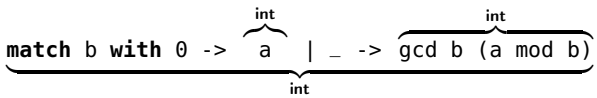
Finally, let us rewrite Euclid's Algorithm from the previous chapter:

```
gcd : int → int → int

let rec gcd a b =
  if b = 0 then a else gcd b (a mod b)
```

Now in pattern matching style:

```
gcd : int → int → int

let rec gcd a b =
  match b with
    0 -> a
  | _ -> gcd b (a mod b)
```

The type of a whole **match** … **with** … expression is determined by the types of the expressions on the right hand side of each -> arrow, all of which must be alike:



We use pattern matching whenever it is easier to read and understand than **if** … **then** … **else** expressions.

Questions

1. Rewrite the `not` function from the previous chapter in pattern matching style.
2. Use pattern matching to write a recursive function which, given a positive integer n, returns the sum of all the integers from 1 to n.
3. Use pattern matching to write a function which, given two numbers x and n, computes x^n.
4. For each of the previous three questions, comment on whether you think it is easier to read the function with or without pattern matching. How might you expect this to change if the functions were much larger?
5. What does `match 1 + 1 with 2 -> match 2 + 2 with 3 -> 4 | 4 -> 5` evaluate to?
6. There is a special pattern `x..y` to denote continuous ranges of characters, for example `'a'..'z'` will match all lowercase letters. Write functions `islower` and `isupper`, each of type **char** $\rightarrow$ **bool**, to decide on the case of a given letter.

So Far

1 Integers `min_int` ... `-3 -2 -1 0 1 2 3` ... `max_int` of type **int**. Booleans `true` and `false` of type **bool**. Characters of type **char** like `'X'` and `'!'`.

Mathematical operators `+ - * / mod` which take two integers and give another.

Operators `= < <= > >= <>` which compare two values and evaluate to either `true` or `false`.

The conditional **`if`** *expression1* **`then`** *expression2* **`else`** *expression3*, where *expresssion1* has type **bool** and *expression2* and *expression3* have the same type as one another.

The boolean operators && and || which allow us to build compound boolean expressions.

2 Assigning a name to the result of evaluating an expression using the **`let`** *name* = *expression* construct. Building compound expressions using **`let`** *name1* = *expression1* **`in`** **`let`** *name2* = *expression2* **`in`** ...

Functions, introduced by **`let`** *name argument1 argument2* ... = *expression*. These have type $\alpha \to \beta$, $\alpha \to \beta \to \gamma$ etc. for some types α, β, γ etc.

Recursive functions, which are introduced in the same way, but using **`let rec`** instead of **`let`**.

3 Matching patterns using **`match`** *expression1* **`with`** *pattern1* `|` ... `->` *expression2* `|` *pattern2* `|` ... `->` *expression3* |... The expressions *expression2*, *expression3* etc. must have the same type as one another, and this is the type of the whole **`match ... with`** expression.