Chapter 3

Programming with Objects in a World

3.1 Construction of new objects

In the previous Section we have encountered some very simple and intuitive datatypes. However, if we really want to compute something with these datatypes, they should be slightly more interesting then fruit. Let's look a little more closely at the Torres game, and reflect on how we can represent the towers and the board.

We need to express that one piece can go on top of another and that knight can be on top of a castle. In ML the solution is that we can simply apply constructors to other objects. It is expressed as follows:

```
datatype Tower
  = Empty
  | Knight of Color
  | Tile of Tower
```

Here are some examples of `Towers`. The first is a tower of height four that has an empty top.

```
Tile (Tile (Tile (Tile Empty)))
```

The next is an empty tower.

```
Empty
```

And last, we have a green knight on top of a two story tower.

```
Tile (Tile (Knight Green))
```

What I haven't told you yet, but which will become clear in the next few lectures is that a constructor can be applied to an object of any arbitrary type.

The next problem is the board, which must represented somehow. The individual towers should be mounted to a grid, otherwise they are somewhat afloat, and we will never be able to write out a function that computes the size of the foundation of a castle. Remember, the scoring is done by multiplying the size with the height of the highest knight in a castle.

Here are a few ideas on how we could represent an 8 by 8 board. The first way of doing it seems awfully verbose. Every individual field is passed as an argument to the constructor of the board `Board1`.

```
datatype Board1
  = Board1 of Tower * Tower * Tower * Tower * Tower * Tower * Tower * Tower
            * Tower * Tower * Tower * Tower * Tower * Tower * Tower * Tower
            * Tower * Tower * Tower * Tower * Tower * Tower * Tower * Tower
            * Tower * Tower * Tower * Tower * Tower * Tower * Tower * Tower
            * Tower * Tower * Tower * Tower * Tower * Tower * Tower * Tower
            * Tower * Tower * Tower * Tower * Tower * Tower * Tower * Tower
            * Tower * Tower * Tower * Tower * Tower * Tower * Tower * Tower
            * Tower * Tower * Tower * Tower * Tower * Tower * Tower * Tower
```

`Tower` in this type is the type of a tower and the `*` stands for a pairing type constructor to be explained in one of the next sections. There is no question about it. This is a very direct encoding. But what renders it inappropriate is the following

- Neighboring fields can hardly be identified,
- Adressing is a problem,
- What's good is that this is always going to be an 8 by 8 field, never more and never less.
- In summary: Nobody would or should use this for representing the board.

Let us abstract the previous idea a little bit, and think of a board as a board of rows. Each row represents eight towers.

```
datatype Board2
  = Row of Tower * Tower * Tower * Tower * Tower * Tower * Tower * Tower
  | Board2 of Board2 * Board2
```

Clearly, this datatype definition does not nearly use as many lines as `Board1`.

- Computing the neighbor relation is still difficult.
- The addressing problem is slightly easier, because we can compute the right line of a board using ML.
- But now not every board that we can represent corresponds to a valid board. (The board could have 10 rows, for example, which is not acceptable.)

Let's try it a different way. The board is 8 by 8, which means that it is squared. If cut in four equal pieces, each piece is still a square. And can be represented as such, recursively!

```
datatype Board
  = Tower of Tower
  | Board of Board * Board
           * Board * Board
```

Let's discuss this version of the board.

- This is a very efficient way of representing such a board
- The neighbor relation is conceptually challenging, but easy to implement (with the right tools you will learn here!)
- Adressing is no problem
- But we have to be careful that our board is not to full. It can now also be that we encounter a degenerate board, which doesn't make too much sense.
- These trees are also called quad trees.

Addressing on such a board can be established by creating explicitly an object of the datatype `Address`, which is defined as follows.

```
datatype Address
  = Here
  | UpperLeft of Address
  | UpperRight of Address
  | LowerLeft of Address
  | LowerRight of Address
```

When you are computing the `Address` of a Tower on a `Board` do the following. When you start, the address is `here`. If you have a board in front you, and the tower you are looking for is in the upper left square, all you have to do is to prefix the address of the tower in that sqaure with the constructor `UpperLeft`. The other three corners of the square can be selected similarly. This might not look easy at first glance, but it is. It is just recursive.

Clearly, there are other ways to address a tower on a board, one is by co-ordinates. The next homework problem gives you the opportunity to elaborate on this idea.

3.2 How to take objects a part

Each object we can form as a member of one of these datatypes is called a *value*. `UpperLeft LowerLeft Here`, for example, is a value of type `address`. In this Section we learn how to work with values and how to take them apart.

As a first example consider a program, or better a function, that cuts off the first element of a non-empty address and returns the tail. We call this function `tail`, and it a function that maps and object of type `Address` into an object of type `Address`. In ML we would write

```
fun tail (UpperLeft A) = A
  | tail (UpperRight A) = A
  | tail (LowerLeft A) = A
  | tail (LowerRight A) = A
```

This is how we define functions. The arguments to `tail` are called patterns. They are essentially values of objects of type `Address`, that contain many variables. Observations:

- The base case is missing, because we required an address to be non-empty.
- Invariants: State clearly what has to hold for the input and output arguments.
- Evaluation order. When ML applies such a function, it tries each of the cases in a top down manner to find one whose pattern fits the given argument. The free variables are then determined, and the body continues executing.

3.3 Recursion

3.4 Some built-in types

3.4.1 Functions

3.4.2 Pairs

3.4.3 Numbers

3.4.4 Booleans

3.5 Reasoning about Objects in the World

3.6 Structuring programs