Chapter 10

New Kinds of Data

So far, we have considered the simple types **int**, **bool**, **char**, the compound type **list**, and tuples. We have built functions from and to these types. It would be possible to encode anything we wanted as lists and tuples of these types, but it would lead to complex and error-strewn programs. It is time to make our own types. New types are introduced using **type**. Here's a type for colours:

OCaml

```
# type colour = Red | Green | Blue | Yellow;;
type colour = Red | Green | Blue | Yellow
```

The name of our new type is colour. It has four *constructors*, written with an initial capital letter: Red, Green, Blue, and Yellow. These are the possible forms a value of type colour may take. Now we can build values of type color:

```
col : colour
cols : colour list
colpair : char × colour

let col = Blue

let cols = [Red; Red; Green; Yellow]

let colpair = ('R', Red)
```

Let us extend our type to include any other colour which can be expressed in the RGB (Red, Green, Blue) colour system (each component ranges from 0 to 255 inclusive, a standard range giving about 16 million different colours).

```
type colour =
  Red
| Green
| Blue
| Yellow
| RGB of int × int × int

cols : colour list

let cols = [Red; Red; Green; Yellow; RGB (150, 0, 255)]
```

We use **of** in our new constructor, to carry information along with values built with it. Here, we are using something of type **int** × **int** × **int**. Notice that the list `cols` of type colour **list** contains varying things, but they are all of the same type, as required by a list. We can write functions by pattern matching over our new type:

```
components : colour → int × int × int

let components c =
  match c with
    Red -> (255, 0, 0)
  | Green -> (0, 255, 0)
  | Blue -> (0, 0, 255)
  | Yellow -> (255, 255, 0)
  | RGB (r, g, b) -> (r, g, b)
```

Types may contain a *type variable* like α to allow the type of part of the new type to vary – i.e. for the type to be polymorphic. For example, here is a type used to hold either nothing, or something of any type:

OCaml

```
# type 'a option = None | Some of 'a;;
type 'a option = None | Some of 'a
```

We can read this as "*a value of type* α **option** *is either nothing, or something of type* α". For example:

```
nothing : α option
number : int option
numbers : int option list
word : char list option

let nothing = None

let number = Some 50

let numbers = [Some 12; None; None; Some 2]

let word = Some ['c'; 'a'; 'k'; 'e']
```

The option type is useful as a more manageable alternative to exceptions where the lack of an answer is a common (rather than genuinely exceptional) occurrence. For example, here is a function to look up a value in a dictionary, returning `None` instead of raising an exception if the value is not found:

```
lookup_opt : α → (α × β) list → β option

let rec lookup_opt x l =
  match l with
    [] -> None
  | (k, v)::t -> if x = k then Some v else lookup_opt x t
```

Now, there is no need to worry about exception handling – we just pattern match on the result of the function.

In addition to being polymorphic, new types may also be recursively defined. We can use this functionality to define our own lists, just like the built-in lists in OCaml but without the special notation:

```
        OCaml

# type 'a sequence = Nil | Cons of 'a * 'a sequence;;
type 'a sequence = Nil | Cons of 'a * 'a sequence
```

We have called our type sequence to avoid confusion. It has two constructors: `Nil` which is equivalent to `[]`, and `Cons` which is equivalent to the `::` operator. `Cons` carries two pieces of data with it – one of type α (the head) and one of type α sequence (the tail). This is the recursive part of our definition. Now we can make our own lists equivalent to OCaml's built-in ones:

Built-in	Ours	Our Type
`[]`	`Nil`	α sequence
`[1]`	`Cons (1, Nil)`	**int** sequence
`['a'; 'x'; 'e']`	`Cons ('a', Cons ('x', Cons ('e', Nil)))`	**char** sequence
`[Red; RGB (20, 20, 20)]`	`Cons (Red, Cons (RGB (20, 20, 20), Nil))`	color sequence

Now you can see why getting at the last element of a list in OCaml is harder than getting at the first element – it is deeper in the structure. Let us compare some functions on OCaml lists with the same ones on our new sequence type. First, the ones for built-in lists.

```
length : α list → int
append : α list → α list → α list

let rec length l =
  match l with
    [] -> 0
  | _::t -> 1 + length t

let rec append a b =
  match a with
    [] -> b
  | h::t -> h :: append t b
```

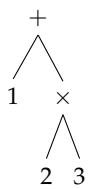
And now the same functions with our new sequence type:

```
length : α sequence → int
append : α sequence → α sequence → α sequence

let rec length s =
  match s with
    Nil -> 0
  | Cons (_, t) -> 1 + length t

let rec append a b =
  match a with
    Nil -> b
  | Cons (h, t) -> Cons (h, append t b)
```

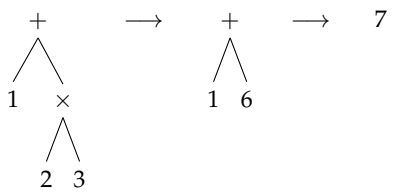
Notice how all the conveniences of pattern matching such as completeness detection and the use of the underscore work for our own types too.

A Type for Mathematical Expressions

Our sequence was an example of a recursively-defined type, which can be processed naturally by recursive functions. Mathematical expressions can be modeled in the same way. For example, the expression $1 + 2 \times 3$ could be drawn like this:



Notice that, in this representation, we never need parentheses – the diagram is unambiguous. We can evaluate the expression by reducing each part in turn:



Here's a suitable type for such expressions:

```
type expr =
  Num of int
| Add of expr * expr
| Subtract of expr * expr
| Multiply of expr * expr
| Divide of expr * expr
```

For example, the expression $1 + 2 * 3$ is represented in this data type as:

```
Add (Num 1, Mul (Num 2, Num 3))
```

We can now write a function to evaluate expressions:

```
evaluate : expr → int

let rec evaluate e =
  match e with
    Num x -> x
  | Add (e, e') -> evaluate e + evaluate e'
  | Subtract (e, e') -> evaluate e - evaluate e'
  | Multiply (e, e') -> evaluate e * evaluate e'
  | Divide (e, e') -> evaluate e / evaluate e'
```

Building our own types leads to clearer programs with more predictable behaviour, and helps us to think about a problem – often the functions are easy to write once we have decided on appropriate types.

Questions

1. Design a new type rect for representing rectangles. Treat squares as a special case.
2. Now write a function of type rect → **int** to calculate the area of a given rect.
3. Write a function which rotates a rect such that it is at least as tall as it is wide.
4. Use this function to write one which, given a rect **list**, returns another such list which has the smallest total width and whose members are sorted widest first.
5. Write `take`, `drop`, and `map` functions for the sequence type.
6. Extend the expr type and the `evaluate` function to allow raising a number to a power.
7. Use the option type to deal with the problem that `Division_by_zero` may be raised from the `evaluate` function.

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.

4 Lists, which are ordered collections of zero or more elements of like type. They are written between square brackets, with elements separated by semicolons e.g. `[1; 2; 3; 4; 5]`. If a list is non-empty, it has a head, which is its first element, and a tail, which is the list composed of the rest of the elements.

The `::` "cons" operator, which adds an element to the front of a list. The `@` "append" operator, which concatenates two lists together.

Lists and the `::` "cons" symbol may be used for pattern matching to distinguish lists of length zero, one, etc. and with particular contents.

5 Matching two or more things at once, using commas to separate as in **`match`** `a, b` **`with`** `0, 0 ->` *expression1* `| x, y ->` *expression2* `|` ...

6 Anonymous functions **`fun`** *name* `->` *expression*. Making operators into functions as in `( < )` and `( + )`.

7 Defining exceptions with **`exception`** *name*. They can carry extra information by adding **`of`** *type*. Raising exceptions with **`raise`**. Handling exceptions with **`try`** ... **`with`** ...

8 Tuples to combine a fixed number of elements `(a, b)`, `(a, b, c)` etc. with types $\alpha \times \beta$, $\alpha \times \beta \times \gamma$ etc.

9 Partial application of functions by giving fewer than the full number of arguments. Partial application with functions built from operators.

10 New types with **`type`** *name* = *constructor1* **`of`** *type1* `|` *constructor2* **`of`** *type2* `|` ... Pattern matching on them as with the built-in types. Polymorphic types.