Pattern-matching

Pattern-matching is a powerful way to do *structural deconstruction* of complex values. A "pattern" is a language construct that can be used on the left-hand side of let expressions, as well as the match expressions that we will see shortly, to suggest the form of the value and to pick out parts of that value.

You have already used pattern-matching without knowing it. For instance when we wrote:

```
let f (x, y) = x + y
f (1, 2)
```

we were actually using pattern-matching. We placed (x, y) in place of the parameter, and what we effectively told OCAML is "I expect this function to be called with the argument being a pair, and I want you to bind x to the first component of the pair and y to the second component when you evaluate the function body". Later on when the call f (1, 2), OCAML will compare the value (1, 2) to the pattern (x, y), it will see that the formats match, and it will bind x to 1 and y to 2.

A similar example would be:

```
let (x, y) = (1, 2) in x + y;;
```

Pattern definition

We will start by describing the forms that a "pattern" can take. We will then discuss how a pattern matches a value. A pattern can be:

- An identifier (x, y etc).
- An underscore _ (called the "catchall").
- A tuple (p1, p2, ..., pn), where each of the p1, p2,... are patterns.
- A literal value, e.g. [], (), 234.1 etc.
- A "list pattern" p1 :: p2, where p1 and p2 are patterns.
- A "constructor pattern". We will see more of these later, but for now this would be either None or Some p where p is another pattern, to match option values.

Here then are some examples of patterns:

```
(_, x :: rest)
(x, (y, z))
(None, x)
(Some (x, y), z)
(Some x) :: rest
```

Pattern matching a value

When the program is evaluated, the question that will arise is whether a pattern `p` matches a value `v`, and what changes to the dynamic environment this would elicit. Here are the rules for that:

- An identifier will match any kind of value, and the environment will be extended to include that binding of the identifier to the value.
- An underscore will match any kind of value, and the environment will not change. The value is effectively ignored.
- A tuple will only match if the corresponding value has the same number of components, and if all the sub-patterns match the corresponding components of the value. The environment will only be extended by whatever bindings were created by the subpatterns.
- A literal-value-pattern will only match the corresponding value if it is in fact the same value. The environment remains unchanged.
- A list pattern `p1 :: p2` will only match a list of the form `v1 :: v2` where the pattern `p1` matches `v1` and the pattern `p2` matches `v2`.
- A "constructor pattern" matches a value if the value is also built by that constructor and the contained pattern matches the corresponding contained value. For example the pattern `Some (x, y)` will match a value `Some (2, 3)` and will bind `x` to `2` and `y` to `3`.

Let us practice this a bit. Which of the following patterns match the corresponding values? What bindings do they produce? After you answer it on the board, test it on the OCAML interpreter by typing `let pattern = value;;`.

1. pattern `(Some _, x)`, value `(Some 3, (1, 2))`.
2. pattern `x :: rest`, value `[1; 2; 3]`.
3. pattern `x :: rest`, value `[]`.
4. pattern `x :: []`, value `[1]`.
5. pattern `x :: y :: []`, value `[(1, 3); (3, 2)]`.
6. pattern `(None, Some x)`, value `(Some 3, Some 2)`.

The `match-with` construct

The match-with construct offers us a powerful way to deconstruct values by using patterns. Its syntax is as follows:

```
match e with
| p1 -> e1
| p2 -> e2
| p3 -> e3
...
```

A match–with expression is evaluated as follows:

- The expression e is evaluated in the current environment and produces a value v.
- The first pattern p1 is compared to the value v. If it matches, then the expression e1 is evaluated in the current environment extended by the bindings created by the matching of p1 to v. This ends the computation and returns the resulting value.
- If the first pattern did not match v, we proceed to try the next pattern, and so on.
- If we arrive at the end and no pattern matched, a runtime error is produced. The typechecker will actually warn you if that is a possibility.

The typechecking is a little more complicated, so we will omit all the details, but the bottom line is that the expressions e1, e2, e3, ..., must all have the same type. Also the type of e must be consistent with the kinds of types that the patterns expect. The system looks at all the patterns and produces one common type from them, then expects e to have that type.

Let us look at some examples:

```
match e with
| Some (x, y) -> x + y
| None -> 0
```

Let us think of what type information is contained in the above code:

- Looking the type of the patterns, we see that they are both option type constructors, so we expect e to be of some option type t option.
- Looking at the first pattern we can see that t must in fact be a pair type, so we are looking at a type like (t1 * t2) option.
- Looking at the first expression (x + y) we see that both x and y must be integers, and so the type for e must in fact be (int * int) option.
- Also the result type of the expression must match the types of both x + y and 0. These are both integer types, and they match, so that is OK.

Here are some examples of what would happen for various values of `e`:

- If `e` is `Some (1, 2)`, then the result will be `1 + 2 = 3`.
- If `e` is `None`, then the result will be `0`.
- If `e` is `(1, 2)`, then we get a typechecking error.
- If `e` is `Some 2`, then we get a typechecking error.

As further practice, think through the type considerations in the following code:

```
match e with
| Some (Some x) -> x
| Some _ -> 1
| None -> 0
```

The importance of pattern-matching will become clear when we combine it with recursion to offer expressive ways of processing functions.