Type Aliases and Type Variants

We continue our exploration of OCAML types with two concepts, type aliases and type variants.

Type Aliases

OCAML allows you to give alternative names to any existing type. This is known as a **type alias**.

As a simple example, suppose are creating some functions for working with integer fractions, like $\frac{5}{4}$. We want to define addition for such things, multiplication etc.

We can represent a fraction as a pair (5, 4). Now we can refer to these in our code via the type int * int, but that type describes pairs of integers in general. In our case these pairs of integers have a further meaning, and our type system doesn't express that.

This is a good use case for a type alias. We can write:

```
type fraction = int * int;;
```

Now from that point on in our code we can use fraction to describe a pair of integers that is meant to be a fraction. For example our definition for fraction multiplication might look like this (we will later do it in a more proper way, reducing the resulting fraction to simple terms):

```
let mult (((a, b), (a', b')) : fraction * fraction) : fraction = (a * a', b * b')
```

It is important to understand that as far as OCAML is concerned, there is no difference between fraction and int * int. So on occasion it might show you int * int instead of fraction, or the other way around. We have already experienced this situation, with the system showing bytes rather than string. This is because there is a built in type alias.

Type Variants

Type variants are a much more powerful language construct. It allows us to construct what is often called a "one-of" type, where a value of this type falls into one of a number of different alternatives. For instance we can define a "number" as something that is an integer or a float. The definition would look like this:

```
type number = IntN of int | FloatN of float
```

The capitalized words "IntN" and "FloatN" are called *constructors* and are used to identify which of the two *variants* a specific value is. Example values of type number would be IntN 5 and FloatN 2.3.

Contrast this with an "each-of" type, namely a pair int * float. In this case to get a value of the new type we had to use one value of each of the two types. In a type variant we only need to use one value from one of the related types.

Variant types don't always need to carry extra information. For example we can define a type that describes the card suits:

```
type suit = Clubs | Spades | Hearts | Diamonds
```

There are exactly 4 different values of type suit, represented by the 4 constructors above. Thing kind of thing, where there is just a series of alternatives that carry no extra information, is often called an *enumeration*.

We utilize a value of a variant type by doing a pattern matching via a match–with expression. For instance we can define a function

```
let is_black (s : suit) : bool =
    match s with
    | Clubs | Spades -> true
    | Hearts | Diamonds -> false
```

As usual, OCAML will tell you if you missed a case, as long as you don't use the catchall pattern (_).

As another example, here is how we can convert a "number" to a float:

```
let to_float (n : number) : float =
    match n with
    | IntN i -> float_of_int i
    | FloatN f -> f

to_float (IntN 4)
to_float (FloatN 2.3)
```

You have already seen variant types before, namely the option and list types. A value of type Option is a variant with two possibilities, None or Some of t where t is the contained type. Similarly for a list.

Self-referential types

Type variants allow types to refer to themselves, allowing us to create some interesting recursive structures (and actually the list type falls into this category). For example we can define a binary tree with integers on the nodes as:

```
type itree = Nil | Node of itree * int * itree
```

So a value of type itree is either Nil a Node containing a triple of another itree (left subtree), an integer (the value at the node), and a another itree (right subtree). A "leaf" would then look like Node (Nil, 4, Nil). Pattern matching allows for some interesting recursive processing of the tree.

Notice this recursive nature of the definition: One of the variants for an itree expects other previously created itrees in it. If we didn't have the Nil variant to get us started, we wouldn't have been able to create any values of this type.

Practice

1. We already have defined a card "suit" type earlier. Define a card value type: The value of a card can be either a "numerical value", which is meant to hold an

integer from 2 to 10 but your type specification will only be able to require that it be integer, or it can hold one of the faces, listed as individual variants Jack, Queen, King, Ace. We do treat the ace as a special case. So your definition should have a total of 5 variants.

2. Define a type alias that defines a card as a pair of a suit and a value.

3. Define a type alias that specifies a hand as a list of cards.

4. Write a function is_valid that takes as input a card and returns a boolean indicating whether it's a valid card.

5. Write a function value that takes as input a card and returns an integer indicating the card's numerical value. Ace counts for 11, numerical values count as themselves and all faces count as 10.

6. Write a function hand_value that takes as input a "hand" and returns the total value of the cards in the hand.

7. Write a function valid_hand that takes as input a "hand" and returns a boolean indicating if it is a "valid hand". A valid hand should consist of exactly 5 valid cards that are distinct. Implement any helper methods you need within the function.