How does OCaml support parametric polymorphism?

Expressions can have types that include type variables, standing for an arbitrary type.

How are type variables written?

Lowercase identifiers starting with a lowercase letter prefixed by an apostrophe.

Add type annotations to a function declaration.

```
let funcname (p1 : pt1) ... (pn : ptn) : rest = ...
```

The parens are essential.

How are tuples written?

And their types?

Comma-separated values, not necessarily surrounded by parens.

The types are written `*`-separated.

What is *destructuring*?

Demonstrate on tuples.

Decomposition of an aggregate type using pattern matching.

```
# let a, b = 1, 2;;
val a : int = 1
val b : int = 2
```

Access the elements of a tuple.

- Use pattern matching.
- For a 2-tuple use `fst` and `snd`.

How are tuples written?

`;`-separated values surrounded by square brackets.

How is the *cons* operator used?

Both to prepend to a list and to pattern match the head of a list.

How are parameterized types written?

With the type parameter preceding the main type.

```
string list
'a list (* polymorphic *)
```

What is an association list?

How is it queried?

A list of pairs, used as a simple associative map.

`List.assoc 'a -> ('a * 'b)`
looks for the associated element or raises `Not_found`.

What is *tail recursion*?

Why is it desirable?

A function is tail recursive if all recursive calls are returned by the function without any additional computation.

Tail recursive functions are preferred over general recursive functions as they can be optimized by the compiler to avoid consuming additional stack space. (They are rewritten as loops).

List accumulator values in recursive functions are best built ...

... in reverse, to benefit from constant-time cons.

The final result will be reversed with `List.rev`.