# Datatypes

## CS3100 Fall 2019

## Review

Previously

- Function definition and application
- Anonymous and recursive functions
- Tail call optimisation

This lecture,

- Data types
- Pattern matching

## Type aliases

OCaml support the definition of aliases for existing types. For example,

In [19]:

```
type int_float_pair = int * float
```

Out[19]:

```
type int_float_pair = int * float
```

In [20]:

```
let x = (10, 3.14)
```

Out[20]:

```
val x : int * float = (10, 3.14)
```

In [21]:

```
let y : int_float_pair = x
```

Out[21]:

```
val y : int_float_pair = (10, 3.14)
```

## Records

- Records in OCaml represent a collection of named elements.
- A simple example is a point record containing x, y and z fields:

In [22]:

```
type point = {
  x : int;
  y : int;
  z : int;
}
```

Out[22]:

```
type point = { x : int; y : int; z : int; }
```

## Records: Creation and access

We can create instances of our point type using `{ ... }`, and access the elements of a point using the `.` operator:

In [23]:

```
let origin = { y = 0; x = 0;z = 0 }

let get_y (r : point) = r.y
```

Out[23]:

```
val origin : point = {x = 0; y = 0; z = 0}
```

Out[23]:

```
val get_y : point -> int = <fun>
```

## Records: Functional update

- New records can also be created from existing records using the `with` keyword.

In [24]:

```
let p = { origin with z = 10 }
```

Out[24]:

```
val p : point = {x = 0; y = 0; z = 10}
```

- `p` is a new record with the same fields as `origin` except `z`.
- `origin` remains unchanged!

In [25]:

```
origin
```

Out[25]:

```
- : point = {x = 0; y = 0; z = 0}
```

## Records: Field punning

Another useful trick with records is field punning, which allows you to replace:

In [26]:

```
let mk_point x y z = { x = x; y = y; z = z }
```

Out[26]:

```
val mk_point : int -> int -> int -> point = <fun>
```

with

In [27]:

```
let mk_point x y z = { x; y; z }
```

Out[27]:

```
val mk_point : int -> int -> int -> point = <fun>
```

## Product Types

- Records and tuples are known as **product types**.
  - Each value of a product type includes all of the types that constitute the product.

```
type person_r = {name: string; age: int; height: float}
type person_t = string * int * float
```

- Records are indexed by *names* whereas *tuples* are indexed by positions (1st, 2nd, etc.).

## what is the sum type?

# VARIANTS

## Defining variants

The type definition syntax is:

```
type t =
| C1 of t1
| C2 of t2
| C3 of t2
| ...
```

- C1, C2, C2 are known as constructors
- t1, t2 and t3 are optional data carried by constructor
- Also known as **Algebraic Data Types**

In [28]:

```
type color =
  | Red
  | Green
  | Blue
```

Out[28]:

```
type color = Red | Green | Blue
```

In [29]:

```
let v = (Green , Red)
```

Out[29]:

```
val v : color * color = (Green, Red)
```

In [30]:

```
type point = {x : int; y : int}

type shape =
  | Circle of point * float (* center, radius *)
  | Rect of point * point   (* lower-left, upper-right *)
  | ColorPoint of point * color
```

Out[30]:

```
type point = { x : int; y : int; }
```

Out[30]:

```
type shape =
    Circle of point * float
  | Rect of point * point
  | ColorPoint of point * color
```

In [31]:

```
Circle ({x=4;y=3}, 2.5)
```

Out[31]:

```
- : shape = Circle ({x = 4; y = 3}, 2.5)
```

In [32]:

```
Rect ({x=3;y=4}, {x=7;y=9})
```

Out[32]:

```
- : shape = Rect ({x = 3; y = 4}, {x = 7; y = 9})
```

# Recursive variant types

Let's define an integer list

In [33]:

```
type intlist =
  | INil
  | ICons of int * intlist
```

Out[33]:

```
type intlist = INil | ICons of int * intlist
```

In [34]:

```
ICons (1, ICons (2, ICons (3, INil)))
```

Out[34]:

```
- : intlist = ICons (1, ICons (2, ICons (3, INil)))
```

- `Nil` and `Cons` originate from Lisp.

## String List

```
type stringlist =
  | SNil
  | Scons of string * stringlist
```

- Now what about `pointlist`, `shapelist`, etc?

## Parameterized Variants

In [35]:

```
type 'a lst =
    Nil
  | Cons of 'a * 'a lst
```

Out[35]:

```
type 'a lst = Nil | Cons of 'a * 'a lst
```

In [36]:

```
Cons (1, Cons (2, Nil))
```

Out[36]:

```
- : int lst = Cons (1, Cons (2, Nil))
```

In [37]:

```
Cons ("Hello", Cons("World", Nil))
```

Out[37]:

```
- : string lst = Cons ("Hello", Cons ("World", Nil))
```

# Type Variable

- **Variable**: name standing for an unknown value
- **Type Variable**: name standing for an unknown type

- Java example is `List<T>`

- OCaml syntax for type variable is a single quote followed by an identifier
  - `'foo`, `'key`, `'value`
- Most often just `'a`, `'b`.
  - Pronounced "alpha", "beta" or "quote a", "quote b".

# Polymorphism

- The type `'a lst` that we had defined earlier is a **polymorphic data type**.
- poly = many, morph = change.
- write functionality that works for many data types.
- Related to Java Generics and C++ template instantiation.
- In `'a lst`, `lst` is known as a **type constructor**.
  - constructs types such as `int lst`, `string lst`, `shape lst`, etc.

# OCaml built-in lists are just variants

OCaml effectively codes up lists as variants:

```
type 'a list = [] | :: of 'a * 'a list
```

- `[]` and `::` are constuctors.
- Just a bit of syntactic magic to use `[]` and `::` as constructors rather than alphanumeric identifiers.

In [38]:

```
[]
```

Out[38]:

```
- : 'a list = []
```

In [39]:

```
1::2::[]
```

Out[39]:

```
- : int list = [1; 2]
```

## Null

"I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object-oriented language. My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. **But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement.** This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years."

**- Sir Tony Hoare**

## Option: A Built-in Variant

- OCaml does not have a null value.

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

In [40]:

```
None
```

Out[40]:

```
- : 'a option = None
```

In [41]:

```
Some 10
```

Out[41]:

```
- : int option = Some 10
```

In [42]:

```
Some "Hello"
```

Out[42]:

```
- : string option = Some "Hello"
```

## When to use option types

```
type student = { name : string; rollno : string;
                 marks : int}
```

- what value will you assign for `marks` field before the exams are taken?
  - `0` is not a good answer since it might also be the case that the student actually scored 0.

```
type student = { name : string; rollno : string;
                 marks : int option }
```

- Use `None` to indicate the exam has not been taken.

## Question

Given records, variants and tuples, which one would you pick for the following cases?

1. Represent currency denominations 10, 20, 50, 100, 200, 500, 2000.
2. Students who have name and roll numbers.
3. A dessert which has a sauce, a creamy component, and a crunchy component.b

- Tuples are convenient for local uses
  - Returning a pair of values
  - Pattern matching multiple things at once.

# Fin.