Exercise

What is the type of the following expression

```
fun x y -> x y;
```

Solution

```
x :'a -> 'b,
y :'a

fun: ('a->'b)->'a->'b
```

Exercise

Write a function that has two arguments

- A number m
- Function f of type `int->real`

The function should compute $\Sigma_{0 \leq i \leq m-1} f(i)$

Solution

```
fun summation f m =
   let fun sum (i,z) : real =
      if i=m then z else sum(i+1,z+(f i))
   in sum (0,0.0)
end;
```

Exceptions

Syntax

```
raise <Exception>
```

Example

```
exception dividebyzero

if a=0 then raise dividebyzero else b/a;
```

Exercise

Write a function that takes as argument a function f and integer value m and computes

$$\mathsf{min}_{0 \leq i \leq m-1} f$$

The function should return an exception if $m = 0$. Use this function to compute

$$\mathsf{min}_{0 \leq i \leq m-1} \mathsf{min}_{0 \leq j \leq n-1} g(i, j)$$

Solution

```
exception Minimum;

fun minimum f 0 = raise Minimum
  | minimum f m =
    let fun min2 (x,y): real = if x<y then x else y
fun minn (i,z) =
            if  i=m  then  z   else  minn (i+1, min2(z, f i))
    in  minn(1, f 0)  end;

fun minimum2 g m n = minimum (fn i=> minimum (fn j => g(i,j)) n) m;
```

Wildcards

- Use wildcard _ to catch exceptions
- Report error message ~1

Modified example

```
type currency = string;
type money = real * currency;

fun convert (amount, to) =
    let val toeur = fn
       (x,"eur") => x
     | (x,"usd") => x / 1.05
     | (x,"ounce_gold") => x * 1113.0
     | (_,_) => ~1.0
   in
     ( case to of
        "eur" => toeur amount
      | "usd" => toeur amount * 1.05
      | "ounce_gold" => toeur amount / 1113.0
      | _ => ~1.0
      , to)
  end;
```

Datatype

- Better to avoid runtime errors and get compilation/static checking errors instead
- Keyword `datatype` defines new types
- `currency` type: exhaustive list

```
datatype currency = eur | usd | ounce_gold;
```

- Try

```
val c = eur;
```

- The values are actually *value constructors* and could have arguments

Using type

- Define `money` datatype ("eur" etc. already used above)

```
datatype money = Eur of real | Usd of real | Ounce_gold of real;
```

- What is `Eur`? Try

```
Eur;
Eur 0.5;
```

Conversion function

```
datatype currency = eur | usd | ounce_gold;
datatype money = Eur of real | Usd of real | Ounce_gold of real;

fun convert (amount, to) =
    let val toeur = fn
       Eur x => x
     | Usd x => x / 1.05
     | Ounce_gold x => x * 1113.0
   in
     case to of
        eur => Eur (toeur amount)
      | usd => Usd (toeur amount * 1.05)
      | ounce_gold => Ounce_gold (toeur amount / 1113.0)
  end;
```

Result of type `money`

```
datatype currency = eur | usd | ounce_gold;
datatype money = Eur of real | Usd of real | Ounce_gold of real;

fun convert (amount, to) =
    let val toeur = fn
       Eur x => x
     | Usd x => x / 1.05
     | Ounce_gold x => x * 1113.0
   in
     ( case to of
        eur => toeur amount
      | usd => toeur amount * 1.05
      | ounce_gold => toeur amount / 1113.0
      , to)
  end;
```

What is the syntax for this function?

Answer

```
convert (Eur 1.1, usd);
```

Operations on types

- Just defining types is not very useful; we must define operations on them as well
- Number: Real or integer

```
datatype number = integer of int | real_number of real;
```

- Operations: Add and subtract numbers

```
val add_numbers = fn (integer a, integer b) => integer (a+b)
  | (integer a, real_number b) => real_number ( (real a) + b)
  | (real_number  a, integer b) => real_number ( a + (real b))
  | (real_number a, real_number b) => real_number ( a + b);

val subtract_numbers = fn (integer a, integer b) => integer (a-b)
  | (integer a, real_number b) => real_number ( (real a) - b)
  | (real_number  a, integer b) => real_number ( a - (real b))
  | (real_number a, real_number b) => real_number ( a - b);
```

Exercise

Define multiplication

Solution

```
val mult_numbers = fn (integer a, integer b) => integer (a*b)
  | (integer a, real_number b) => real_number ( (real a) * b)
  | (real_number  a, integer b) => real_number ( a * (real b))
  | (real_number a, real_number b) => real_number ( a * b);
```

Recursive types

- T can be part of the definition of T
- Example: List of integers

```
datatype list = leaf of int | node of (int * list);
```

- Operation: Calculate length

```
val rec Len = fn leaf _ => 1
     | node (_,l) => 1 + Len l;
```

- Concatenate

```
val rec Concat  = fn (leaf v,l) => node(v,l)
   | (node(v,l1),l2) => node (v,Concat (l1,l2));
```

- More on recursive types later

Type variables

- Support "parametric polymorphism" (to be studied next week)

```
fn x => x;
```

- Result uses a *type variable*. In the theory part we shall use greek letters, so this function has type $\alpha \to \alpha$. It maps the argument, of any type, to the result, of the *same type*

```
val it = fn: 'a -> 'a
```

- Type variables can be used to define parametrized types

```
datatype 'a oerror = error | value of 'a;
```

- For example

```
error;
value 5;
```

- Data type can have type variables as arguments

```
datatype ('a,'b)  strange_type = none | one of 'a | two of 'a*'b;
```

Modular programming: Structure and signature

- Structure: Group set of definitions in an environment to avoid variable name clashes (similar to C++ namespace)
- Signature: To create a software interface
- Structude definition

```
structure circle = struct
  val pi = 3.14152;
  val circle_area = fn r => r * r * pi;
end;
```

- Note the signature
- Try

```
circle_area 2;
```

- Why doesn't this work?

Structures

- Not visible outside
- Use

```
circle.circle_area 3.0;
```

Sets: Without structures

```
val emptyset = [];

val rec isin =
  fn x =>
    (fn [] => false
    | y::l =>
      if (x=y) then
         true
      else
       isin x l);

val addin =
    fn x =>
      fn l =>
       if (isin x l)
          then
            l
       else
           x::l;
```

Continued

```
val rec removefrom
    = fn x =>
         (fn [] => []
          | y::l =>
            if (x=y) then
                l
           else
     y::(removefrom x l));
```