Pattern Matching

- Case split based on the **structure** of datatypes:

```
case ⟨exp⟩ of
   ⟨pattern⟩  =>  ⟨exp⟩
 | ⟨pattern⟩  =>  ⟨exp⟩
 |    ...     ...   ...
 | ⟨pattern⟩  =>  ⟨exp⟩
```

- **First** matching rule is used (only)
- Parts can be captured and **named**:

```
case aList of
     nil      =>   0
 |  (x ::  t)  =>  x+1
```

- **Literals** can be used in patterns:

```
case anInt of
   42  =>   0
 |  x   =>  2*x
```

- **Exception** if no pattern matches (and warning at compile-time)
- Special syntax for **function definition**:

```
fun  name  ⟨pattern⟩ = ···
  |  name  ⟨pattern⟩ = ···
  |   ...
```

- Special syntax: `as`:

```
case ⟨exp⟩ of
 (⟨variable⟩ as ⟨pattern⟩) => ···
| ...
```

Example

```
case [1,2] of
  (l as x::t) => ···
```

`l=[1,2]` `x=1` `t=[2]`

Pattern Matching in Function Definition

```
- fun fact x =
= case x of
= 0 => 1
= | x => x * fact (x-1);
val fact = fn : int -> int

- fun fact 0 = 1
= | fact n = n * fact (n-1);
val fact = fn : int -> int

- fun len nil = 0
= | len (x::t) = 1 + len t;
val len = fn : 'a list -> int

- fun len [] = 0
= | len (_::t) = 1 + len t;
val len = fn : 'a list -> int

- fun len (_::l) = 1 + len l
= | len _ = 0;
val len = fn : 'a list -> int

- fun len l =
= case l of
= [] => 0
= | (_::t) => 1 + len t;
val len = fn : 'a list -> int
```

```
- fun moreThanOne (_::_::_) = true
= | moreThanOne _ = false;
val moreThanOne = fn : 'a list -> bool

- fun append (l1, l2) =
= if null l1 then l2 else hd l1 :: append (tl l1, l2);
val append = fn : 'a list * 'a list -> 'a list

- fun append ([], l) = l
=   |  append (x :: t, l) = x :: append (t, l);
val append = fn : 'a list * 'a list -> 'a list

- fun nth (l, 0) = hd l
= | nth (l, n) = nth (tl l, n-1);
val nth = fn : 'a list * int -> 'a

- fun concat ([]) = []
= | concat (h :: t) = append (h, concat (t));
val concat = fn : 'a list list -> 'a list

- fun revh ([], r) = r
=   |  revh (x :: t, r) = revh (t, x :: r);
val revh = fn : 'a list * 'a list -> 'a list
- fun rev l = revh (l, []);
val rev = fn : 'a list -> 'a list
```

```
- fun fill (_, 0) = []
=  |  fill (x, n) = x :: fill (x, n-1);
val fill = fn : 'a * int -> 'a list
- fill ("X", 10);
val it = ["X","X","X","X","X","X","X","X","X","X"] : string list

- fun filter (low, high, x :: t) =
=            (if low <= x andalso x <= high then [x] else []) @ filter (low, high, t)
=  |  filter (_, _, []) = [];
val filter = fn : int * int * int list -> int list
- filter (3, 5, [1,5,6,2,7,2,3,4]);
val it = [5,3,4] : int list

- fun genericFilter (_, []) = []
=  |  genericFilter (test, x :: l) =
=     if test x then x :: genericFilter (test, l) else genericFilter (test, l);
val genericFilter = fn : ('a -> bool) * 'a list -> 'a list

- fun filter (low, high, l) = genericFilter (fn x => low <= x andalso x <= high, l);
val filter = fn : int * int * int list -> int list
- filter (3, 5, [1,5,6,2,7,2,3,4]);
val it = [5,3,4] : int list

fun add_or_sub f =
    if f 1
    then fn x => x+1
    else fn x => x-1
val add = add_or_sub (fn x => x = 1)
val sub = add_or_sub (fn x => x = 2) 3
```

"Curried" Functions

```
- fun find0 [] = false
=   |  find0 (x :: t) = x=0 orelse find0 t;
val find0 = fn : int list -> bool

- find0 [1,2,3];
val it = false : bool
- find0 [1,2,0,3];
val it = true : bool

- fun findPair (_, []) = false
=   |  findPair (v, (x :: t)) = x=v orelse findPair (v, t);
val findPair = fn : ''a * ''a list -> bool

- findPair (3, [1,2,3]);
val it = true : bool
- findPair (4, [1,2,3]);
val it = false : bool
```

```
- fun find v = fn l => findPair (v, l);
val find = fn : ''a -> ''a list -> bool

- find 3;
-  val it = fn : int list -> bool
- find "A";
- val it = fn : string list -> bool
- find 3 [1,2,3];
val it = true : bool
- find 4 [1,2,3];
val it = false : bool

- fun find _ [] = false
=  |  find v (x :: t) = x=v orelse find v t;
val find = fn : ''a -> ''a list -> bool

- find 1;
val it = fn : int list -> bool
- find 1 [1,2,3];
val it = true : bool
- find 1 [];
val it = false : bool
- val find0 = find 0;
val find0 = fn : int list -> bool
- find0 [1,2,3];
val it = false : bool
- find0 [1,2,0];
val it = true : bool
- val f = find "A";
val f = fn : string list -> bool
```

```
- fun powerFind _ [] = false
=   |  powerFind test (x :: l) = test x orelse powerFind test l;
val powerFind = fn : ('a -> bool) -> 'a list -> bool
- fun gt3 x = x > 3;
val gt3 = fn : int -> bool
- powerFind gt3 [1,2,3];
val it = false : bool
- powerFind gt3 [1,2,3,4];
val it = true : bool
- powerFind (fn x => x > 3) [1,2,3,4];
val it = true : bool
- fun find v = powerFind (fn x => x=v); (* this is find from before *)
val find = fn : ''a -> ''a list -> bool
- val find0 = powerFind (fn x => x=0); (* this is find0 from before *)
val find0 = fn : int list -> bool
```

```
- fun sumh ([], a) = a
=   |   sumh (x :: t, a) = sumh (t, x+a);
val sumh = fn : int list * int -> int
- fun sum l = sumh (l, 0);
val sum = fn : int list -> int
- sum [1,2,3];
val it = 6 : int
```

```
- fun sumh ([], a) = a
=   |   sumh (x :: t, a) = sumh (t, x+a);
val sumh = fn : int list * int -> int
- fun sum l = sumh (l, 0);
val sum = fn : int list -> int
- sum [1,2,3];
val it = 6 : int

- fun sumh a [] = a
=   |   sumh a (x :: l) = sumh (x+a) l;
val sumh = fn : int -> int list -> int
- val sum = sumh 0;
val sum = fn : int list -> int
- sum [1,2,3];
val it = 6 : int
```