

# Higher-Order Programming

From our discussions of data types we that types such as

```
int → int
int * float → bool
char list → int
```

all describe sets of functions – but a data type is a set of data values.

☞ We can treat functions like data values that are members of a type.

## Example:

```
- floor;
val it = fn : real -> int
- val x = floor;
val x = fn : real -> int
```

# Higher-Order Programming

Def: In higher-order programming functions wither take functions as parameters or return functions as return values.

Example: A generic type conversion function from real to int – this functions takes a real value and a specific type conversion function as arguments and converts the value according to the specific conversion function.

```
- fun genconv (x:real, f:real -> int) = f(x);
val genconv = fn : real * (real -> int) -> int
```

Specific conversion functions:

```
floor: real → int
ceil:  real → int
round: real → int
```

```
- genconv(3.2, floor);
val it = 3 : int
```

```
- genconv(3.2, ceil);
val it = 4 : int
```

```
- genconv(3.2, round);
val it = 3 : int
```

# Anonymous Functions

Sometimes functions are too simple to warrant a full fledged function definition – ML provides something called anonymous function definitions for building small functions on the fly.

## Syntax:

<anonymous-function> ::= **fn** <pattern> => <expression>

<pattern> ::= any valid ML pattern

<expression> ::= an valid ML expression

Examples: a simple increment by one function

```
- fn x => x + 1;  
val it = fn : int -> int
```

```
- (fn x => x+1) 1;  
val it = 2 : int
```

# Anonymous Functions

Why do we bother with anonymous functions?

They are a great way to help us write generic code which then can be made to do specific things via anonymous functions.

Example: a generic increment function.

```
- fun geninc (a, f) = f a;  
val geninc = fn : 'a * ('a -> 'b) -> 'b
```

```
- geninc (2, (fn x => x + 3));  
val it = 5 : int
```

```
- geninc (2, fn x => x + 1);  
val it = 3 : int
```

# Function Currying

Multi-parameter functions are written as a cascade of anonymous functions.

Example:

```
- fun sum (a,b) = a + b;
```

```
val sum = fn : int * int -> int
```

```
- fun csum a = (fn b => a + b);
```

```
val csum = fn : int -> int -> int
```

Currying has ramifications on how you call functions:

```
- sum (1,2);
```

```
val it = 3 : int
```

BUT

```
- csum 1 2; ← no tuples!
```

```
val it = 3 : int
```

# Function Currying

A "Curried" function with two arguments is the composition of a named function with an anonymous function.

Example:

```
- fun csum a = (fn b => a + b);
```

anonymous function

named function 'csum'

Example: partial evaluation

```
- val p = csum 1;
```

```
val p = fn : int -> int
```

```
- p 2;
```

```
val it = 2 : int
```

```
p ≡ (fn b => 1 + b)
```

partially evaluated function!

# Function Currying

Example: a function that adds three numbers.

```
- fun cadd3 a = fn b => fn c => a + b + c;  
val cadd3 = fn : int -> int -> int -> int
```

type of 1<sup>st</sup> input argument

```
- cadd3 (1,2,3);  
ERROR....
```

tuple int\*int\*int; incorrect type for 1<sup>st</sup> argument