**Lecture 33:**

- OCaml: Tuples, More functions

**Announcements:**

- HW-6 due
- HW-7 out

## OCaml Tuples

**A "tuple" is a fixed size collection of values**

- Each tuple value can have a **different** type
- Tuple values and types are denoted using parentheses ...

```
# (1, 2) ;;                     (* int pair *)
- : int * int = (1, 2)
# ('a', true) ;;                (* heterogeneous pair *)
- : char * bool = ('a', true)
# (1, 2., 'a') ;;               (* heterogenous 3-tuple *)
- : int * float * char = (1, 2., 'a')
```

- The "*" is pronounced "*cross*" (think of it as "*and*")

**Lists and tuples can be nested** ... but watch out for types!

```
# ([1; 2], ['a'; 'b']) ;;
- : int list * char list = ([1; 2], ['a'; 'b'])
# [(1, 2); ('a', 'b')] ;;
Error: This expression has type char but an expression
       was expected of type int
```

# OCaml Tuples

### “Pairs” (2-tuples)

- Can access elements using `fst` and `snd` functions

```
# fst ;;
- : 'a * 'b -> 'a = <fun>
# snd ;;
- : 'a * 'b -> 'b = <fun>
# (1, "foo") ;;
- : int * string = (1, "foo")
# fst (1, "foo")
- : int = 1
# snd (1, "foo")
- : string = "foo"
```

- Note this only works with pairs (2-tuples)!

# OCaml Basics

### A more general approach: “Pattern Matching” (first look)

Using `fst` and `snd` to define a function:

```
# let pair_add_1 p = ((fst p) + 1, (snd p) + 1) ;;
val pair_add_1 : int * int -> int * int = <fun>

# pair_add_1 (1,2) ;;
- : int * int = (2, 3)
```

Alternatively, by “matching” on the (sub) structure of pairs:

```
# let pair_add_1 (x, y) = (x + 1, y + 1) ;;
val pair_add_1 : int * int -> int * int = <fun>

# pair_add_1 (2, 3) ;;
- : int * int = (3, 4)
```

- Q: how does OCaml figure out the function types here?

# OCaml Recursive Functions

### Defining Recursive Functions in OCaml

First (wrong) attempt ...

```
# let fac n = if n <= 1 then 1 else fac (n-1) * n ;;
Error: Unbound value fac
```

Second (correct) attempt ... use the `rec` modifier (for recursive)

```
# let rec fac n = if n <= 1 then 1 else fac (n-1) * n ;;
val fac : int -> int = <fun>
# fac 10 ;;
- : int = 3628800
```

# OCaml Recursive Functions

Defining mutually recursive functions:

- E.g., one function $f$ calls $g$, and $g$ calls $f$
- We can use and to define them in the same `let` binding

```
# let rec f n =
    if n < 0 then g n else n + 1
  and g n =
    if n >= 0 then f n else n - 1
  ;;
val f : int -> int = <fun>
val g : int -> int = <fun>
# f 1 ;;
- : int = 2
# f (-1) ;;
- : int = -2
# g 1 ;;
- : int = 2
```

## OCaml Exceptions

### Basic Exceptions for Error Cases

- OCaml supports exceptions and exception handling
- Generate “failure” exceptions with `failwith` ...

```
let rec fac n =
  if n = 0 then 1
  else if n > 0 then n * fac (n-1)
  else failwith "Negative Value"
;;

# fac (-1) ;;
Exception: Failure "Negative Value".

# failwith ;;
- : string -> 'a = <fun>
```

- Note `failwith` returns a value of any type!

## OCaml List Functions

**The classic “head” (first elem) function**: ... aka “car”

```
# List.hd [4; 1; 5] ;;
- : int = 4

# List.hd [] ;;
Exception: Failure "hd".

# List.hd ;;
- : 'a list -> 'a
```

**Can define using pattern matching**: ... more later

```
let head xs =
  match xs with
    | [] -> failwith "Empty list"
    | x::t -> x                                (* better w/ wildcards: x::_ *)
```

- Two cases for `xs`: either empty or `x` plus rest
- Using cons to “<u>**deconstruct**</u>” the list ... `[]`, `x::t` are the patterns

# OCaml List Functions

**The classic "tail" function**: ... aka "cdr"

```
# List.tl [4; 1; 5] ;;
- : int list = [1; 5]

# List.tl [1] ;;
- : int list = []

# List.tl [] ;;
Exception: Failure "tl".

# List.tl ;;
- : 'a list -> 'a list = <fun>
```

**Can define using pattern matching**:

```
let tail xs =
  match xs with
    | [] -> failwith "Empty list"
    | _::t -> t
```

# OCaml List Functions

**Head and tail functions useful for defining other functions**

```
let empty xs = xs == []

(* length: 'a list -> int *)
let rec length xs =
  if empty xs then 0 else 1 + length (tail xs)

(* member: 'a -> 'a list -> bool *)
let rec member x xs =
  if empty xs then false
  else if head xs == x then true
  else member x (tail xs)
```

- alternatively: `List.is_empty`, `List.length`, `List.mem`
- all of these can be defined using pattern matching instead (more later)
- Can add type info: `let rec member (x:  'a) (xs:  'a list) :  bool = ...`