CSC 330 / Spring 2026

OCaml

Functions, Compound Data Types

Luke Kuligowicz

Ibrahim Numanagić /nu-ma-nagg-ich/

based on Dan Grossman's lecture materials

Reminder: syntax & semantics

- Syntax: how programs are written down
  - Roughly “spelling and grammar”
- Semantics: what programs actually mean
  - Compile time (static) semantics: type checking
  - Run time (dynamic) semantics: evaluation
- Both syntax and semantics matter
  - Semantics is the essence of a programming language
  - Syntax is more about taste

Functions

Functions

- **The most important** building block in the whole course
  - Functions let us abstract some computation by parameterizing over parts of an expression
  - Like Python functions or Java methods, have arguments and result
    - But no classes, this/self, returns (!), etc.
- Example:
  - **function binding**

```
let max ((x : int), (y : int)) =
  if x > y then x else y
```

  - **function call**

```
max (3, 17)
```

Example with recursion

```
let rec pow ((x : int), (y : int)) =
  if y = 0 then 1 else x * pow (x, y - 1)

let cube (x : int) =
  pow (x, 3)

let sixtyfour = cube 4
let fortytwo = pow (2, 4) + pow (4, 2) + cube 2 + 2
```

Heads up: recursion!

- If you're not comfortable with recursion yet, you will be soon 😉
  - We'll write **many** recursive functions in lecture and homework
  - Mostly over recursive data structures like lists (next lecture)
- Makes sense because recursive calls solve *simpler* problems
- Recursion is more general than loops!
  - We won't write a single loop in OCaml (!)
  - Easy to simulate loops with recursion
  - Loops often (not always) obscure simple, elegant solutions

Questions

- Three questions for a new language construct:

  syntax, typing rules, evaluation rules

- We have two new language constructs:

  function bindings, function calls

Functions: bindings

- **Syntax**:

```
let f ((x1 : t1), ..., (xN : tN)) = e
let rec f ((x1 : t1), ..., (xN : tN)) = e
```

  where:

  - `f, x1, ..., xN` are variable names
  - `t1, ..., tN` are types
  - `e` is an expression

Meta-syntax

- We use `...` to indicate some syntax is optional
  - Example: `rec`
  - So the syntax for function bindings can be written as:

    ```
    let [rec] f ((x1 : t1), ..., (xN : tN)) = e
    ```

- But the presence/absence of `rec` **does** matter
  - It affects the typing rules and evaluation rules!
  - Just saying it is correct function-binding syntax with or without it

Functions: bindings

This is a simplified-for-now syntax story:

1. Can write the return type if you want (example code won't do this):

   ```
   let [rec] f ((x1 : t1), ..., (xN : tN)) : t = e
   ```

2. Can omit argument types (and we will *after* A1):

   ```
   let [rec] f (x1, ..., xN) = e
   ```

3. Will see a different, more common, way to do multiple arguments later

⚠️ **Note**: Be careful on argument syntax—slight errors can produce strange error messages (or even worse, behaviour)!

Functions: bindings

- **Type checking**:

  `let [rec] f ((x1 : t1), ..., (xN : tN)) = e`

  – If `e` has type `t` in the static environment containing:

    1. The *enclosing* (up to this point) static environment
    2. `x1` $\mapsto$ `t1`, …, `xN` $\mapsto$ `tN` arguments and their types
    3. If `rec` is given, `f` $\mapsto$ `t1 * ... * tN -> t`:
       - then: add `f` $\mapsto$ `t1 * ... * tN -> t` to the static environment
       - else: report error and fail

Function types

- New kind of type! `f` $\mapsto$ `t1 * ... * tN -> t`
  - Argument types `t1, ..., tN` separated by `*`
  - Result (return) type `t` after the arrow
- How this is used for typechecking function bindings:
  - `f` has this type in the rest of program (not in earlier bindings!)
  - Arguments available in `e`, but not after (unsurprising)
  - Calling `f` evaluates `e`, so return type of `f` is type of `e`
  - Magic for now (will study it in a few weeks): typechecker figures out `t` even when `f` is recursive

Functions: bindings

- **Evaluation**:

  `let [rec] f ((x1 : t1), ..., (xN : tN)) = e`

  - Nothing to do! Function are values!
    - Real story a bit more nuanced, but we will get to it…
  - Add `f` to the dynamic environment so later expressions can call
    - `f` is bound to the function being defined here
    - And for recursion, if `rec` is present, then `f` also bound within `e`

Functions: calling

- **Syntax:**

  `f (e1, ..., eN)`

  where `f, e1, ..., eN` are expressions (will generalize this a bit later)

- **Type checking:**
  - If:
    - `f` has type `f : t1 * ... * tN -> t`, and
    - `e1` has type `t1`, and …, and `eN` has type `tN`
  - then: `f (e1, ..., eN)` has type `t`

Functions: calling

- **Evaluation**:

  `e0 (e1, ..., eN)`

  1. Evaluate `e0` in current dynamic environment
     - Since `e0` type checked, the result will be a function
     - So the result is some

       `let [rec] f ((x1 : t1), ..., (xN : tN)) = e`
  2. In current dynamic environment, evaluate: `e1` to value `v1`, … , `eN` to value `vN`
  3. The `e0 (e1, ..., eN)` call now evaluates to the result of evaluating `e` in an *extended dynamic environment* with `x1 ↦ v1, ..., xN ↦ vN`

> **Technically**: *extend dynamic environment where* `f` *was defined*. But more on this later!

Heads up: function gotchas

- Be careful with argument declaration and argument passing
  - Else confusing error messages
- **Remember**: cannot refer to later bindings
  - So helper functions must come first
  - Mutual recursion is handled specially

Demonstration 1: Functions

Demonstration 1: Functions

```
# let max ((x : int), (y : int)) =
  if x > y then x else y ;;
```

Demonstration 1: Functions

```
# let max ((x : int), (y : int)) =
  if x > y then x else y ;;
val max : int * int -> int = <fun>
```

Demonstration 1: Functions

```
# let max ((x : int), (y : int)) =
  if x > y then x else y ;;
val max : int * int -> int = <fun>

# max (3, 17) ;;
```

Demonstration 1: Functions

```
# let max ((x : int), (y : int)) =
  if x > y then x else y ;;
val max : int * int -> int = <fun>

# max (3, 17) ;;
- : int = 17
```

Demonstration 1: Functions

```
# let max ((x : int), (y : int)) =
  if x > y then x else y ;;
val max : int * int -> int = <fun>

# max (3, 17) ;;
- : int = 17

# (* Note: pow assumes y >= 0 *)
let rec pow ((x : int), (y : int)) =
  if y = 0 then 1 else x * pow (x, y - 1)
let cube (x : int) = pow (x, 3) ;;
```

Demonstration 1: Functions

```
# let max ((x : int), (y : int)) =
  if x > y then x else y ;;
val max : int * int -> int = <fun>

# max (3, 17) ;;
- : int = 17

# (* Note: pow assumes y >= 0 *)
let rec pow ((x : int), (y : int)) =
  if y = 0 then 1 else x * pow (x, y - 1)
let cube (x : int) = pow (x, 3) ;;
val pow : int * int -> int = <fun>
val cube : int -> int = <fun>
```

Demonstration 1: Functions

```
# let max ((x : int), (y : int)) =
  if x > y then x else y ;;
val max : int * int -> int = <fun>

# max (3, 17) ;;
- : int = 17

# (* Note: pow assumes y >= 0 *)
let rec pow ((x : int), (y : int)) =
  if y = 0 then 1 else x * pow (x, y - 1)
let cube (x : int) = pow (x, 3) ;;
val pow : int * int -> int = <fun>
val cube : int -> int = <fun>

# let sixtyfour = cube 4
let fortytwo = pow (2, 4) + pow (4, 2) + cube 2 + 2 ;;
```

Demonstration 1: Functions

```
# let max ((x : int), (y : int)) =
  if x > y then x else y ;;
val max : int * int -> int = <fun>

# max (3, 17) ;;
- : int = 17

# (* Note: pow assumes y >= 0 *)
let rec pow ((x : int), (y : int)) =
  if y = 0 then 1 else x * pow (x, y - 1)
let cube (x : int) = pow (x, 3) ;;
val pow : int * int -> int = <fun>
val cube : int -> int = <fun>

# let sixtyfour = cube 4
let fortytwo = pow (2, 4) + pow (4, 2) + cube 2 + 2 ;;
val sixtyfour : int = 64
val fortytwo : int = 42
```

Demonstration 1: Functions

Demonstration 1: Functions

```
# let is_even (x : int) =
    (x mod 2) = 0

  let average ((x : int), (y : int)) =
    (x + y) / 2

  let average_ceil ((x : int), (y : int)) =
    ((x + y) / 2) + (if not (is_even (x + y)) then 1 else 0) ;;
```

Demonstration 1: Functions

```
# let is_even (x : int) =
  (x mod 2) = 0

let average ((x : int), (y : int)) =
  (x + y) / 2

let average_ceil ((x : int), (y : int)) =
  ((x + y) / 2) + (if not (is_even (x + y)) then 1 else 0) ;;
val is_even : int -> bool = <fun>
val average : int * int -> int = <fun>
val average_ceil : int * int -> int = <fun>
```

Demonstration 1: Functions

```
# let is_even (x : int) =
  (x mod 2) = 0

let average ((x : int), (y : int)) =
  (x + y) / 2

let average_ceil ((x : int), (y : int)) =
  ((x + y) / 2) + (if not (is_even (x + y)) then 1 else 0) ;;
val is_even : int -> bool = <fun>
val average : int * int -> int = <fun>
val average_ceil : int * int -> int = <fun>

# average (3, 4)
average_ceil (3, 4) ;;
```

Demonstration 1: Functions

```
# let is_even (x : int) =
  (x mod 2) = 0

let average ((x : int), (y : int)) =
  (x + y) / 2

let average_ceil ((x : int), (y : int)) =
  ((x + y) / 2) + (if not (is_even (x + y)) then 1 else 0) ;;
val is_even : int -> bool = <fun>
val average : int * int -> int = <fun>
val average_ceil : int * int -> int = <fun>

# average (3, 4)
average_ceil (3, 4) ;;
- : int = 3
- : int = 4
```

Demonstration 1: Functions

```
# let is_even (x : int) =
  (x mod 2) = 0

let average ((x : int), (y : int)) =
  (x + y) / 2

let average_ceil ((x : int), (y : int)) =
  ((x + y) / 2) + (if not (is_even (x + y)) then 1 else 0) ;;
val is_even : int -> bool = <fun>
val average : int * int -> int = <fun>
val average_ceil : int * int -> int = <fun>

# average (3, 4)
average_ceil (3, 4) ;;
- : int = 3
- : int = 4

# (* More than two arguments are possible *)
let average_ceil_or_not ((x : int), (y : int), (ceil : bool)) =
  ((x + y) / 2) + (if ceil && not (is_even (x + y)) then 1 else 0) ;;
average_ceil_or_not (3, 4, true) ;;
average_ceil_or_not (3, 4, false) ;;
```

Demonstration 1: Functions

```
# let is_even (x : int) =
  (x mod 2) = 0

let average ((x : int), (y : int)) =
  (x + y) / 2

let average_ceil ((x : int), (y : int)) =
  ((x + y) / 2) + (if not (is_even (x + y)) then 1 else 0) ;;
val is_even : int -> bool = <fun>
val average : int * int -> int = <fun>
val average_ceil : int * int -> int = <fun>

# average (3, 4)
average_ceil (3, 4) ;;
- : int = 3
- : int = 4

# (* More than two arguments are possible *)
let average_ceil_or_not ((x : int), (y : int), (ceil : bool)) =
  ((x + y) / 2) + (if ceil && not (is_even (x + y)) then 1 else 0) ;;
average_ceil_or_not (3, 4, true) ;;
average_ceil_or_not (3, 4, false) ;;
val average_ceil_or_not : int * int * bool -> int = <fun>
- : int = 4
- : int = 3
```

Demonstration 1: Functions

```
# let is_even (x : int) =
  (x mod 2) = 0

let average ((x : int), (y : int)) =
  (x + y) / 2

let average_ceil ((x : int), (y : int)) =
  ((x + y) / 2) + (if not (is_even (x + y)) then 1 else 0) ;;
val is_even : int -> bool = <fun>
val average : int * int -> int = <fun>
val average_ceil : int * int -> int = <fun>

# average (3, 4)
average_ceil (3, 4) ;;
- : int = 3
- : int = 4

# (* More than two arguments are possible *)
let average_ceil_or_not ((x : int), (y : int), (ceil : bool)) =
  ((x + y) / 2) + (if ceil && not (is_even (x + y)) then 1 else 0) ;;
average_ceil_or_not (3, 4, true) ;;
average_ceil_or_not (3, 4, false) ;;
val average_ceil_or_not : int * int * bool -> int = <fun>
- : int = 4
- : int = 3

# (* There is no automatic int<->float casting *)
let average_f ((x : float) , (y : float)) = (x +. y) /. 2.0
average_f (3., 4.) ;;
```

Demonstration 1: Functions

```
# let is_even (x : int) =
  (x mod 2) = 0

let average ((x : int), (y : int)) =
  (x + y) / 2

let average_ceil ((x : int), (y : int)) =
  ((x + y) / 2) + (if not (is_even (x + y)) then 1 else 0) ;;
val is_even : int -> bool = <fun>
val average : int * int -> int = <fun>
val average_ceil : int * int -> int = <fun>

# average (3, 4)
average_ceil (3, 4) ;;
- : int = 3
- : int = 4

# (* More than two arguments are possible *)
let average_ceil_or_not ((x : int), (y : int), (ceil : bool)) =
  ((x + y) / 2) + (if ceil && not (is_even (x + y)) then 1 else 0) ;;
average_ceil_or_not (3, 4, true) ;;
average_ceil_or_not (3, 4, false) ;;
val average_ceil_or_not : int * int * bool -> int = <fun>
- : int = 4
- : int = 3

# (* There is no automatic int<->float casting *)
let average_f ((x : float) , (y : float)) = (x +. y) /. 2.0
average_f (3., 4.) ;;
val average_f : float * float -> float = <fun>
- : float = 3.5
```

Demonstration 1: Functions

```
# let is_even (x : int) =
  (x mod 2) = 0

let average ((x : int), (y : int)) =
  (x + y) / 2

let average_ceil ((x : int), (y : int)) =
  ((x + y) / 2) + (if not (is_even (x + y)) then 1 else 0) ;;
val is_even : int -> bool = <fun>
val average : int * int -> int = <fun>
val average_ceil : int * int -> int = <fun>

# average (3, 4)
average_ceil (3, 4) ;;
- : int = 3
- : int = 4

# (* More than two arguments are possible *)
let average_ceil_or_not ((x : int), (y : int), (ceil : bool)) =
  ((x + y) / 2) + (if ceil && not (is_even (x + y)) then 1 else 0) ;;
average_ceil_or_not (3, 4, true) ;;
average_ceil_or_not (3, 4, false) ;;
val average_ceil_or_not : int * int * bool -> int = <fun>
- : int = 4
- : int = 3

# (* There is no automatic int<->float casting *)
let average_f ((x : float) , (y : float)) = (x +. y) /. 2.0
average_f (3., 4.) ;;
val average_f : float * float -> float = <fun>
- : float = 3.5

# (* There is no automatic int<->float casting *)
average_f (3, 4) ;;
```

Demonstration 1: Functions

```
# let is_even (x : int) =
  (x mod 2) = 0

let average ((x : int), (y : int)) =
  (x + y) / 2

let average_ceil ((x : int), (y : int)) =
  ((x + y) / 2) + (if not (is_even (x + y)) then 1 else 0) ;;
val is_even : int -> bool = <fun>
val average : int * int -> int = <fun>
val average_ceil : int * int -> int = <fun>

# average (3, 4)
average_ceil (3, 4) ;;
- : int = 3
- : int = 4

# (* More than two arguments are possible *)
let average_ceil_or_not ((x : int), (y : int), (ceil : bool)) =
  ((x + y) / 2) + (if ceil && not (is_even (x + y)) then 1 else 0) ;;
average_ceil_or_not (3, 4, true) ;;
average_ceil_or_not (3, 4, false) ;;
val average_ceil_or_not : int * int * bool -> int = <fun>
- : int = 4
- : int = 3

# (* There is no automatic int<->float casting *)
let average_f ((x : float) , (y : float)) = (x +. y) /. 2.0
average_f (3., 4.) ;;
val average_f : float * float -> float = <fun>
- : float = 3.5

# (* There is no automatic int<->float casting *)
average_f (3, 4) ;;
Error: This expression has type int but an expression was expected of type
         float
  Hint: Did you mean `3.'?
```

So much power in these questions

- Three questions per language construct:

  syntax, typing rules, evaluation rules

- Two new language constructs:

  function bindings, function calls

This is the most important building block in the entire course!

```
let rec pow ((x : int), (y : int)) =
  if y = 0
  then 1
  else x * pow (x, y - 1)
let cube (x : int) =
    pow (x, 3)
let sixtyfour = cube 4
```

Tuples and lists

Compound data types

- So far: integers, booleans, variables, conditionals, functions
- **Now**: two ways to build data with multiple parts:
  - [ ] tuples: fixed number of "pieces" with possibly different types
  - [ ] lists: any number of "pieces" all with the same type
- Later: more general ways to create compound data (trees, maps, graphs, etc.)

General design principle

- Whenever we design a new type `t` we need two things:
  - A way to build (introduce, construct) values of type `t`
  - A way to use (eliminate, destruct) values of type `t`
- **Examples**:
  - Build `t1 -> t2` with function bindings, use with function calls
  - Build `bool` with `true` and `false`, use with `if e1 then e2 else e3`

Pairs (2-tuples): building

- **Syntax**: `(e1, e2)` where `e1` and `e2` are expressions
- **Type checking**:
  - If `e1` has type `t1` and `e2` has type `t2`:
    - then: `(e1, e2)` has type `t1 * t2`
    - else: report error and fail (only if `e1` or `e2` do not type check)
- **Evaluation**:
  - If `e1` evaluates to value `v1` and `e2` evaluates to value `v2`:
    - then: `(e1, e2)` evaluates to `(v1, v2)`

The star of the show

- `*` is a type constructor: it builds tuple types out of other types
  - So `int * bool`, `string * int`, `int * (int * int)`, are 3 of the infinite number of types we can construct with `*`
- Tuple types are also known as product types
- But the connection to multiplication is obscure and best ignored
  - Just using the same character in unrelated syntax
  - `3 * 4` is an expression that evaluates to 12, while `int * int` is a type

Nested pairs

- Pairs can be nested
- Nesting is *not* a new feature!
- Simply a consequence of the syntax and semantics rules we set up `((1, 4), (2, (true, 3))):` `(int * int) * (int * (bool * int))`
- Compositionality is a hallmark of good design
  - Comes from orthogonality: one thing does not affect another

Pairs (2-tuples): using

- **Syntax**: `fst e` and `snd e` where `e` is an expression
- **Type checking**:
  - If `e` has type `t1 * t2`:
    - then: `fst e` has type `t1` and `snd e` has type `t2`
- **Evaluation**:
  - If `e` evaluates to a pair of values `(v1, v2)`:
    - then: `fst e` evaluates to `v1` and `snd e` evaluates to `v2`

⚠️: `fst` and `snd` are actually library functions, but we will pretend they are “built in” for another week or so

Tuples: building

- **Syntax**: `(e1, ..., eN)` where `e1, ..., eN` are expressions and $N \geq 2$
  - This *is* a new feature: `(5, true, 7)` is a 3-tuple (triple), **not** any sort of nested pair
- **Type checking**:
  - If `e1` has type `t1`, and …, and `eN` has type `tN`:
    - then: `(e1, ..., eN)` has type `t1 * ... * tN`
    - else: report error and fail
- **Evaluation**:
  - If `e1` evaluates to value `v1`, and …, and `eN` evaluates to value `vN`:
    - then: `(e1, ..., eN)` evaluates to `(v1, ..., vN)`

Tuples: using

- OCaml does not have a built-in way to **use** arbitrary tuples… 🤯
- Instead, OCaml relies on a more general mechanism we will learn soon: pattern-matching
- But A1 uses triples a lot
  - So we wrote `fst3`, `snd3`, and `thd3` for you to treat-like-primitives
  - So `fst` and `snd` for pairs; `fst3`, `snd3`, and `thd3` for triples
  - Why can't `fst` be used for triples? *Type systems have limitations!*

Nested pairs vs. tuples

- OCaml didn't *need* tuples, but they can be convenient
- Parentheses matters in tuple types and tuple expressions!

```
let x = (5, 7, 9)
(* x : int * int * int *)
let seven = snd3 x
(* snd x doesn't typecheck *)
```

```
let x = (5, (7, 9))
(* x : int * (int * int) *)
let seven = fst (snd x)
let pr = snd x
(* snd3 x doesn't typecheck *)
```

Demonstration 2: Tuples

Demonstration 2: Tuples

```
# let swap (pr : int * int) = (snd pr, fst pr)

let sum_pairs ((pr1 : int * int), (pr2 : int * int)) =
  (fst pr1 + fst pr2, snd pr1 + snd pr2)

let four_six = sum_pairs ((1, 2), (3, 4)) ;;
```

Demonstration 2: Tuples

```
# let swap (pr : int * int) = (snd pr, fst pr)

let sum_pairs ((pr1 : int * int), (pr2 : int * int)) =
  (fst pr1 + fst pr2, snd pr1 + snd pr2)

let four_six = sum_pairs ((1, 2), (3, 4)) ;;
val swap : int * int -> int * int = <fun>
val sum_pairs : (int * int) * (int * int) -> int * int = <fun>
val four_six : int * int = (4, 6)
```

Demonstration 2: Tuples

```
# let swap (pr : int * int) = (snd pr, fst pr)

let sum_pairs ((pr1 : int * int), (pr2 : int * int)) =
  (fst pr1 + fst pr2, snd pr1 + snd pr2)

let four_six = sum_pairs ((1, 2), (3, 4)) ;;
 val swap : int * int -> int * int = <fun>
 val sum_pairs : (int * int) * (int * int) -> int * int = <fun>
 val four_six : int * int = (4, 6)

 # val sort_pair : int * int -> int * int = <fun>
val one_four : int * int = (1, 4)
val two_three : int * int = (2, 3) ;;
```

Demonstration 2: Tuples

```
# let swap (pr : int * int) = (snd pr, fst pr)

let sum_pairs ((pr1 : int * int), (pr2 : int * int)) =
  (fst pr1 + fst pr2, snd pr1 + snd pr2)

let four_six = sum_pairs ((1, 2), (3, 4)) ;;
 val swap : int * int -> int * int = <fun>
 val sum_pairs : (int * int) * (int * int) -> int * int = <fun>
 val four_six : int * int = (4, 6)

# val sort_pair : int * int -> int * int = <fun>
val one_four : int * int = (1, 4)
val two_three : int * int = (2, 3) ;;
 val sort_pair : int * int -> int * int = <fun>
 val one_four : int * int = (1, 4)
 val two_three : int * int = (2, 3)
```

Demonstration 2: Tuples

```
# let swap (pr : int * int) = (snd pr, fst pr)

let sum_pairs ((pr1 : int * int), (pr2 : int * int)) =
  (fst pr1 + fst pr2, snd pr1 + snd pr2)

let four_six = sum_pairs ((1, 2), (3, 4)) ;;
 val swap : int * int -> int * int = <fun>
 val sum_pairs : (int * int) * (int * int) -> int * int = <fun>
 val four_six : int * int = (4, 6)

# val sort_pair : int * int -> int * int = <fun>
val one_four : int * int = (1, 4)
val two_three : int * int = (2, 3) ;;
 val sort_pair : int * int -> int * int = <fun>
 val one_four : int * int = (1, 4)
 val two_three : int * int = (2, 3)

# let div_mod ((x : int), (y : int)) = (x / y, x mod y)
let r = div_mod (16, 3) ;;
```

Demonstration 2: Tuples

```
# let swap (pr : int * int) = (snd pr, fst pr)

let sum_pairs ((pr1 : int * int), (pr2 : int * int)) =
  (fst pr1 + fst pr2, snd pr1 + snd pr2)

let four_six = sum_pairs ((1, 2), (3, 4)) ;;
 val swap : int * int -> int * int = <fun>
 val sum_pairs : (int * int) * (int * int) -> int * int = <fun>
 val four_six : int * int = (4, 6)

# val sort_pair : int * int -> int * int = <fun>
val one_four : int * int = (1, 4)
val two_three : int * int = (2, 3) ;;
 val sort_pair : int * int -> int * int = <fun>
 val one_four : int * int = (1, 4)
 val two_three : int * int = (2, 3)

# let div_mod ((x : int), (y : int)) = (x / y, x mod y)
let r = div_mod (16, 3) ;;
 val div_mod : int * int -> int * int = <fun>
 val r : int * int = (5, 1)
```

Demonstration 2: Tuples

```
# let swap (pr : int * int) = (snd pr, fst pr)

let sum_pairs ((pr1 : int * int), (pr2 : int * int)) =
  (fst pr1 + fst pr2, snd pr1 + snd pr2)

let four_six = sum_pairs ((1, 2), (3, 4)) ;;
val swap : int * int -> int * int = <fun>
val sum_pairs : (int * int) * (int * int) -> int * int = <fun>
val four_six : int * int = (4, 6)

# val sort_pair : int * int -> int * int = <fun>
val one_four : int * int = (1, 4)
val two_three : int * int = (2, 3) ;;
val sort_pair : int * int -> int * int = <fun>
val one_four : int * int = (1, 4)
val two_three : int * int = (2, 3)

# let div_mod ((x : int), (y : int)) = (x / y, x mod y)
let r = div_mod (16, 3) ;;
val div_mod : int * int -> int * int = <fun>
val r : int * int = (5, 1)

# (* nesting *)
let middle_elt (prpr : int * (int * int)) =
  fst (snd prpr)
let two = middle_elt (1, (2, 3)) ;;
```

Demonstration 2: Tuples

```
# let swap (pr : int * int) = (snd pr, fst pr)

let sum_pairs ((pr1 : int * int), (pr2 : int * int)) =
  (fst pr1 + fst pr2, snd pr1 + snd pr2)

let four_six = sum_pairs ((1, 2), (3, 4)) ;;
 val swap : int * int -> int * int = <fun>
 val sum_pairs : (int * int) * (int * int) -> int * int = <fun>
 val four_six : int * int = (4, 6)

# val sort_pair : int * int -> int * int = <fun>
val one_four : int * int = (1, 4)
val two_three : int * int = (2, 3) ;;
 val sort_pair : int * int -> int * int = <fun>
 val one_four : int * int = (1, 4)
 val two_three : int * int = (2, 3)

# let div_mod ((x : int), (y : int)) = (x / y, x mod y)
let r = div_mod (16, 3) ;;
 val div_mod : int * int -> int * int = <fun>
 val r : int * int = (5, 1)

# (* nesting *)
let middle_elt (prpr : int * (int * int)) =
  fst (snd prpr)
let two = middle_elt (1, (2, 3)) ;;
 val middle_elt : int * (int * int) -> int = <fun>
 val two : int = 2
```

Demonstration 2: Tuples

Demonstration 2: Tuples

```
 # (* arbitrary tuples *)
let person1 = (("Zachary", "L.", "Tatlock"), 201)
let person2 = (("Daniel", "J.", "Grossman"), 309) ;;
```

Demonstration 2: Tuples

```
# (* arbitrary tuples *)
let person1 = (("Zachary", "L.", "Tatlock"), 201)
let person2 = (("Daniel", "J.", "Grossman"), 309) ;;

val person1 : (string * string * string) * int =
  (("Zachary", "L.", "Tatlock"), 201)
val person2 : (string * string * string) * int =
  (("Daniel", "J.", "Grossman"), 309)
```

Demonstration 2: Tuples

```
# (* arbitrary tuples *)
let person1 = (("Zachary", "L.", "Tatlock"), 201)
let person2 = (("Daniel", "J.", "Grossman"), 309) ;;
 val person1 : (string * string * string) * int =
   (("Zachary", "L.", "Tatlock"), 201)
 val person2 : (string * string * string) * int =
   (("Daniel", "J.", "Grossman"), 309)

# (* ⚠ BUT WAIT! ⚠
 * Not only isn't there a function to get the third (or fourth etc) element,
 * but fst and snd don't work on 3-tuples (or 4-tuples etc.)
 * Indeed, sometimes type systems "get in our way":
 * fst and snd are functions that require pairs (2-tuples)
 * OCaml's answer is pattern-matching (next week) so for A1,
 * we "give you" fst3, snd3, and thd3, which you will use for triples (3-tuples) *)
(* you cannot understand these implementations yet but you can use them *)
let fst3 (x, _, _) = x (* gets the first element of a triple *)
let snd3 (_, x, _) = x (* gets the second element of a triple *)
let thd3 (_, _, x) = x (* gets the third element of a triple *) ;;
```

Demonstration 2: Tuples

```
# (* arbitrary tuples *)
let person1 = (("Zachary", "L.", "Tatlock"), 201)
let person2 = (("Daniel", "J.", "Grossman"), 309) ;;
 val person1 : (string * string * string) * int =
   (("Zachary", "L.", "Tatlock"), 201)
 val person2 : (string * string * string) * int =
   (("Daniel", "J.", "Grossman"), 309)

# (* ⚠️ BUT WAIT! ⚠️
 * Not only isn't there a function to get the third (or fourth etc) element,
 * but fst and snd don't work on 3-tuples (or 4-tuples etc.)
 * Indeed, sometimes type systems "get in our way":
 * fst and snd are functions that require pairs (2-tuples)
 * OCaml's answer is pattern-matching (next week) so for A1,
 * we "give you" fst3, snd3, and thd3, which you will use for triples (3-tuples) *)
(* you cannot understand these implementations yet but you can use them *)
let fst3 (x, _, _) = x (* gets the first element of a triple *)
let snd3 (_, x, _) = x (* gets the second element of a triple *)
let thd3 (_, _, x) = x (* gets the third element of a triple *) ;;
 val fst3 : 'a * 'b * 'c -> 'a = <fun>
 val snd3 : 'a * 'b * 'c -> 'b = <fun>
 val thd3 : 'a * 'b * 'c -> 'c = <fun>
 val middle_initials : string * string = ("L.", "J.")
```

Compound data types

- So far: integers, booleans, variables, conditionals, functions
- Now: two ways to build data with multiple parts:
  - [x] tuples: fixed number of "pieces" with possibly different types
  - [ ] lists: any number of "pieces" all with the same type
- Later: more general ways to create compound data (trees, maps, graphs, etc.)

Lists: building

- **Syntax**: `[e1; ...; eN]` where `e1, ..., eN` are expressions
- **Type checking**:
  - If `e1` has type `t`, and …, and `eN` has type `t`:
    - then: `[e1; ...; eN]` has type `t list`
      - Another type constructor (`list`): type of lists whose elements have *same* type `t`
        - Examples: `bool list int list`, `(int * (int * int)) list`, `int list list` etc.
    - else: report error and fail
- **Evaluation**:
  - If `e1` evaluates to value `v1`, and …, and `eN` evaluates to value `vN`:
    - then: `[e1; ...; eN]` evaluates to `[v1; ...; vN]`

⚠️: Elements are separated by semicolons! Otherwise you can get very weird error messages.
📝: There is no equivalent of Python's `[1, "s", Foo()]` in OCaml

Lists: empty lists

- N can be zero: `[]` is the empty list for any type (`'a list`)
- Recall:
  - If `e1` has type `t`, and …, and `eN` has type `t`, then `[e1; ...; eN]` has type `t list`
- Even if __all __ zero elements of `[]` have type `t`, then `[]` has type `t list`
  - `[]` can be of type `int list`, `bool list`, `int list list`, `(bool * int) list` etc.
  - OCaml writes *any type of list* as `'a list` (or `'b list`, `'c list` etc.)
  - But all list elements of any list have the same type
- These are **polymorphic types**: we will see them a lot in OCaml

Lists: building with `::`

Create (or cons, from *construct*) a list with one more element at the front

- **Syntax**: `e1 :: e2` where `e1` and `e2` are expressions
- **Type checking**:
  - If `e1` has type `t` and `e2` has type `t list`:
    - then: `e1 :: e2` has type `t list`
    - else: report error and fail
- **Evaluation**:
  - If `e1` evaluates to `v1` and `e2` evaluates to `[v2; ...; vN]`:
    - then: `e1 :: e2` evaluates to `[v1; v2; ...; vN]`

Two ways to build?

- `::` is more common in code
- `::` and `[]` is all we *need*
  - `e1 :: e2 :: e3 :: ... :: eN :: []` is the same as `[e1; e2; e3; ...; eN]`
  - First of many examples of **syntactic sugar**
- List values are printed as `[v1; v2; v3; ...; vN]`

⚠️: OCaml lists are (singly) linked lists under the hood

Lists: using

- For now, we will *use* lists with these features:
  - Test if a list is empty with `e = []`
  - Get the **first** element of a *non-empty list* with function `List.hd`
  - Get the **rest** (tail) of a *non-empty list* with function `List.tl` (a list with everything after the first element)
- `List.hd` and `List.tl` will raise an exception if given an empty list
- Later, we will use *pattern matching* to do this more safely and elegantly

Lists: testing for empty list

- **Syntax**: `e = []` where `e` is an expression
- **Type checking**:
  - If has `e` has type `t list`:
    - then: `e = []` has type `bool`
    - else: report error and fail
- **Evaluation**:
  - If `e` evaluates to value `[]`:
    - then: `e = []` evaluates to `true`
    - else: `e = []` evaluates to `false`

Lists: getting the first element

- **Syntax**: `List.hd e` where `e` is an expression
- **Type checking**:
  - If `e` has type `t list`
    - then: `List.hd e` has type `t`
    - else: report error and fail

> **Alternate definition**: `List.hd` has type `'a list -> 'a`

- **Evaluation**:
  - If `e` evaluates to value `[v1; v2; ...; vN]`
    - then: `List.hd e` evaluates to value `v1`
    - else: `e` evaluates to `[]`, so `List.hd e` raises an exception

Lists: getting the tail (everything except the first element)

- **Syntax**: `List.tl e` where `e` is an expression
- **Type checking**:
  - If `e` has type `t list`:
    - then: `List.tl e` has type `t list`
    - else: report error and fail

> **Alternate definition**: `List.tl` has type `'a list -> 'a list`

- **Evaluation**:
  - If `e` evaluates to value `[v1; v2; ...; vN]`:
    - then: `List.tl e` evaluates to value `[v2; ...; vN]`
    - else: `e` evaluates to `[]`, so `List.tl e` raises an exception

Recursion again!

- Functions over lists are usually recursive
  - Only way to get to all the elements
- **Design recipe**: answer the following two questions
  - *What should the answer be for an empty list?*
    - Often thinking about the return type provides a good hint (base case)
  - *What should the answer be for a non-empty list?*
    - Typically this will be in terms of the answer for the tail (recursion)

Lists of tuples

- Processing lists of tuples is a common pattern (especially in A1)
  - Requires *no new features!*
  - Just compose what we've already learned
  - **Again**: compositionality is a hallmark of good design
    - often stemming from orthogonality

Demonstration 3: Lists

Demonstration 3: Lists

```
# (* building lists:
   * from [] (proncounced empty) and :: (pronounced cons); or
   * the [x1; ...; xN] shorthand *)
let empty = []
let some_years = 2019 :: (2020 :: (2021 :: []))
let some_years = 2019 :: 2020 :: 2021 :: []
let some_years = [2019; 2020; 2021] ;;
```

Demonstration 3: Lists

```
# (* building lists:
   * from [] (proncounced empty) and :: (pronounced cons); or
   * the [x1; ...; xN] shorthand *)
let empty = []
let some_years = 2019 :: (2020 :: (2021 :: []))
let some_years = 2019 :: 2020 :: 2021 :: []
let some_years = [2019; 2020; 2021] ;;
val empty : 'a list = []
val some_years : int list = [2019; 2020; 2021]
```

Demonstration 3: Lists

```
# (* building lists:
   * from [] (proncounced empty) and :: (pronounced cons); or
   * the [x1; ...; xN] shorthand *)
let empty = []
let some_years = 2019 :: (2020 :: (2021 :: []))
let some_years = 2019 :: 2020 :: 2021 :: []
let some_years = [2019; 2020; 2021] ;;
val empty : 'a list = []
val some_years : int list = [2019; 2020; 2021]

# (* confusing error message because this is a one-element list of a triple *)
let bad_some_years = [2019,2020,2021] ;;
```

Demonstration 3: Lists

```
# (* building lists:
   * from [] (proncounced empty) and :: (pronounced cons); or
   * the [x1; ...; xN] shorthand *)
let empty = []
let some_years = 2019 :: (2020 :: (2021 :: []))
let some_years = 2019 :: 2020 :: 2021 :: []
let some_years = [2019; 2020; 2021] ;;
val empty : 'a list = []
val some_years : int list = [2019; 2020; 2021]

# (* confusing error message because this is a one-element list of a triple *)
let bad_some_years = [2019,2020,2021] ;;
val bad_some_years : (int * int * int) list = [(2019, 2020, 2021)]
```

Demonstration 3: Lists

```
# (* building lists:
   * from [] (proncounced empty) and :: (pronounced cons); or
   * the [x1; ...; xN] shorthand *)
let empty = []
let some_years = 2019 :: (2020 :: (2021 :: []))
let some_years = 2019 :: 2020 :: 2021 :: []
let some_years = [2019; 2020; 2021] ;;
 val empty : 'a list = []
 val some_years : int list = [2019; 2020; 2021]

# (* confusing error message because this is a one-element list of a triple *)
let bad_some_years = [2019,2020,2021] ;;
 val bad_some_years : (int * int * int) list = [(2019, 2020, 2021)]

# (* naturally, we can make lists using any expressions *)
let more_years = (2020 - 3) :: (fst (sum_pairs (one_four, two_three)))
                            :: (if 0 > 2 then -3 else 4) :: some_years ;;
```

Demonstration 3: Lists

```
# (* building lists:
   * from [] (proncounced empty) and :: (pronounced cons); or
   * the [x1; ...; xN] shorthand *)
let empty = []
let some_years = 2019 :: (2020 :: (2021 :: []))
let some_years = 2019 :: 2020 :: 2021 :: []
let some_years = [2019; 2020; 2021] ;;
val empty : 'a list = []
val some_years : int list = [2019; 2020; 2021]

# (* confusing error message because this is a one-element list of a triple *)
let bad_some_years = [2019,2020,2021] ;;
val bad_some_years : (int * int * int) list = [(2019, 2020, 2021)]

# (* naturally, we can make lists using any expressions *)
let more_years = (2020 - 3) :: (fst (sum_pairs (one_four, two_three)))
                            :: (if 0 > 2 then -3 else 4) :: some_years ;;
val more_years : int list = [2017; 3; 4; 2019; 2020; 2021]
```

Demonstration 3: Lists

Demonstration 3: Lists

```
# (* Because lists typically have any length, we use recursion to make them or process them
   * check for empty: lst = []
   * get first element: List.hd (pronounced head)
   * get the list containing all but the first element: List.tl (pronounced tail) *)

let rec countdown (n : int) =
  if n <= 0 then []
  else n :: countdown (n - 1)

let rec sum_list (xs : int list) =
  if xs = [] then 0
  else (List.hd xs) + sum_list (List.tl xs)

let rec length (xs : 'a list) =
  if xs = [] then 0 else 1 + length (List.tl xs)

let rec nth ((xs: 'a list), (n : int)) =
  if n = 0 then List.hd xs
  else nth (List.tl xs, n - 1) ;;
```

Demonstration 3: Lists

```
# (* Because lists typically have any length, we use recursion to make them or process them
   * check for empty: lst = []
   * get first element: List.hd (pronounced head)
   * get the list containing all but the first element: List.tl (pronounced tail) *)

let rec countdown (n : int) =
  if n <= 0 then []
  else n :: countdown (n - 1)

let rec sum_list (xs : int list) =
  if xs = [] then 0
  else (List.hd xs) + sum_list (List.tl xs)

let rec length (xs : 'a list) =
  if xs = [] then 0 else 1 + length (List.tl xs)

let rec nth ((xs: 'a list), (n : int)) =
  if n = 0 then List.hd xs
  else nth (List.tl xs, n - 1) ;;
val countdown : int -> int list = <fun>
val sum_list : int list -> int = <fun>
val length : 'a list -> int = <fun>
val nth : 'a list * int -> 'a = <fun>
```

Demonstration 3: Lists

```
# (* Because lists typically have any length, we use recursion to make them or process them
   * check for empty: lst = []
   * get first element: List.hd (pronounced head)
   * get the list containing all but the first element: List.tl (pronounced tail) *)

let rec countdown (n : int) =
  if n <= 0 then []
  else n :: countdown (n - 1)

let rec sum_list (xs : int list) =
  if xs = [] then 0
  else (List.hd xs) + sum_list (List.tl xs)

let rec length (xs : 'a list) =
  if xs = [] then 0 else 1 + length (List.tl xs)

let rec nth ((xs: 'a list), (n : int)) =
  if n = 0 then List.hd xs
  else nth (List.tl xs, n - 1) ;;
val countdown : int -> int list = <fun>
val sum_list : int list -> int = <fun>
val length : 'a list -> int = <fun>
val nth : 'a list * int -> 'a = <fun>

# let l = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
let l' = countdown 10
let s = (sum_list l) + (sum_list l')
let len = length l
let fifth = nth (l', 5)
let l = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
let l' = countdown 10
let s = (sum_list l) + (sum_list l')
let len = length l
let fifth = nth (l', 5) ;;
```

Demonstration 3: Lists

```
# (* Because lists typically have any length, we use recursion to make them or process them
   * check for empty: lst = []
   * get first element: List.hd (pronounced head)
   * get the list containing all but the first element: List.tl (pronounced tail) *)

let rec countdown (n : int) =
  if n <= 0 then []
  else n :: countdown (n - 1)

let rec sum_list (xs : int list) =
  if xs = [] then 0
  else (List.hd xs) + sum_list (List.tl xs)

let rec length (xs : 'a list) =
  if xs = [] then 0 else 1 + length (List.tl xs)

let rec nth ((xs: 'a list), (n : int)) =
  if n = 0 then List.hd xs
  else nth (List.tl xs, n - 1) ;;
val countdown : int -> int list = <fun>
val sum_list : int list -> int = <fun>
val length : 'a list -> int = <fun>
val nth : 'a list * int -> 'a = <fun>

# let l = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
let l' = countdown 10
let s = (sum_list l) + (sum_list l')
let len = length l
let fifth = nth (l', 5)
let l = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
let l' = countdown 10
let s = (sum_list l) + (sum_list l')
let len = length l
let fifth = nth (l', 5) ;;
val l : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
val l' : int list = [10; 9; 8; 7; 6; 5; 4; 3; 2; 1]
val s : int = 110
val len : int = 10
val fifth : int = 5
```

Demonstration 3: Lists

```
# (* Because lists typically have any length, we use recursion to make them or process them
   * check for empty: lst = []
   * get first element: List.hd (pronounced head)
   * get the list containing all but the first element: List.tl (pronounced tail) *)

let rec countdown (n : int) =
  if n <= 0 then []
  else n :: countdown (n - 1)

let rec sum_list (xs : int list) =
  if xs = [] then 0
  else (List.hd xs) + sum_list (List.tl xs)

let rec length (xs : 'a list) =
  if xs = [] then 0 else 1 + length (List.tl xs)

let rec nth ((xs: 'a list), (n : int)) =
  if n = 0 then List.hd xs
  else nth (List.tl xs, n - 1) ;;
val countdown : int -> int list = <fun>
val sum_list : int list -> int = <fun>
val length : 'a list -> int = <fun>
val nth : 'a list * int -> 'a = <fun>

# let l = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
let l' = countdown 10
let s = (sum_list l) + (sum_list l')
let len = length l
let fifth = nth (l', 5)
let l = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
let l' = countdown 10
let s = (sum_list l) + (sum_list l')
let len = length l
let fifth = nth (l', 5) ;;
val l : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
val l' : int list = [10; 9; 8; 7; 6; 5; 4; 3; 2; 1]
val s : int = 110
val len : int = 10
val fifth : int = 5

# (* naturally, we can make lists using any expressions *)
let more_years = (2020 - 3) :: (fst (sum_pairs (one_four, two_three)))
                            :: (if 0 > 2 then -3 else 4) :: some_years ;;
```

Demonstration 3: Lists

```
# (* Because lists typically have any length, we use recursion to make them or process them
   * check for empty: lst = []
   * get first element: List.hd (pronounced head)
   * get the list containing all but the first element: List.tl (pronounced tail) *)

let rec countdown (n : int) =
  if n <= 0 then []
  else n :: countdown (n - 1)

let rec sum_list (xs : int list) =
  if xs = [] then 0
  else (List.hd xs) + sum_list (List.tl xs)

let rec length (xs : 'a list) =
  if xs = [] then 0 else 1 + length (List.tl xs)

let rec nth ((xs: 'a list), (n : int)) =
  if n = 0 then List.hd xs
  else nth (List.tl xs, n - 1) ;;
val countdown : int -> int list = <fun>
val sum_list : int list -> int = <fun>
val length : 'a list -> int = <fun>
val nth : 'a list * int -> 'a = <fun>

# let l = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
let l' = countdown 10
let s = (sum_list l) + (sum_list l')
let len = length l
let fifth = nth (l', 5)
let l = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
let l' = countdown 10
let s = (sum_list l) + (sum_list l')
let len = length l
let fifth = nth (l', 5) ;;
val l : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
val l' : int list = [10; 9; 8; 7; 6; 5; 4; 3; 2; 1]
val s : int = 110
val len : int = 10
val fifth : int = 5

# (* naturally, we can make lists using any expressions *)
let more_years = (2020 - 3) :: (fst (sum_pairs (one_four, two_three)))
                            :: (if 0 > 2 then -3 else 4) :: some_years ;;
val more_years : int list = [2017; 3; 4; 2019; 2020; 2021]
```

Demonstration 3: Lists

Demonstration 3: Lists

```
# let rec sum_pair_list (prs : (int * int) list) =
  if prs = [] then 0
  else (fst (List.hd prs)) + (snd (List.hd prs)) + sum_pair_list (List.tl prs)

let rec firsts (prs : ('a * 'b) list) =
  if prs = [] then []
  else (fst (List.hd prs)) :: (firsts (List.tl prs))

let rec seconds (prs : ('a * 'b) list) =
  if prs = [] then []
  else (snd (List.hd prs)) :: (seconds (List.tl prs))

let sum_pair_list2 (prs : (int * int) list) =
  (sum_list (firsts prs)) + (sum_list (seconds prs)) ;;
```

Demonstration 3: Lists

```
# let rec sum_pair_list (prs : (int * int) list) =
  if prs = [] then 0
  else (fst (List.hd prs)) + (snd (List.hd prs)) + sum_pair_list (List.tl prs)

let rec firsts (prs : ('a * 'b) list) =
  if prs = [] then []
  else (fst (List.hd prs)) :: (firsts (List.tl prs))

let rec seconds (prs : ('a * 'b) list) =
  if prs = [] then []
  else (snd (List.hd prs)) :: (seconds (List.tl prs))

let sum_pair_list2 (prs : (int * int) list) =
  (sum_list (firsts prs)) + (sum_list (seconds prs)) ;;
val sum_pair_list : (int * int) list -> int = <fun>
val firsts : ('a * 'b) list -> 'a list = <fun>
val seconds : ('a * 'b) list -> 'b list = <fun>
val sum_pair_list2 : (int * int) list -> int = <fun>
```

Demonstration 3: Lists

```
# let rec sum_pair_list (prs : (int * int) list) =
   if prs = [] then 0
   else (fst (List.hd prs)) + (snd (List.hd prs)) + sum_pair_list (List.tl prs)

  let rec firsts (prs : ('a * 'b) list) =
   if prs = [] then []
   else (fst (List.hd prs)) :: (firsts (List.tl prs))

  let rec seconds (prs : ('a * 'b) list) =
   if prs = [] then []
   else (snd (List.hd prs)) :: (seconds (List.tl prs))

  let sum_pair_list2 (prs : (int * int) list) =
   (sum_list (firsts prs)) + (sum_list (seconds prs)) ;;
val sum_pair_list : (int * int) list -> int = <fun>
val firsts : ('a * 'b) list -> 'a list = <fun>
val seconds : ('a * 'b) list -> 'b list = <fun>
val sum_pair_list2 : (int * int) list -> int = <fun>

# let l = [(1, "a"); (2, "b"); (3, "c")]
  let f = firsts l
  let s = seconds l

  let l = [(1, 2); (3, 4); (5, 6)]
  let f = sum_pair_list l
  let s = sum_pair_list2 l ;;
```

Demonstration 3: Lists

```
# let rec sum_pair_list (prs : (int * int) list) =
  if prs = [] then 0
  else (fst (List.hd prs)) + (snd (List.hd prs)) + sum_pair_list (List.tl prs)

let rec firsts (prs : ('a * 'b) list) =
  if prs = [] then []
  else (fst (List.hd prs)) :: (firsts (List.tl prs))

let rec seconds (prs : ('a * 'b) list) =
  if prs = [] then []
  else (snd (List.hd prs)) :: (seconds (List.tl prs))

let sum_pair_list2 (prs : (int * int) list) =
  (sum_list (firsts prs)) + (sum_list (seconds prs)) ;;
 val sum_pair_list : (int * int) list -> int = <fun>
 val firsts : ('a * 'b) list -> 'a list = <fun>
 val seconds : ('a * 'b) list -> 'b list = <fun>
 val sum_pair_list2 : (int * int) list -> int = <fun>

# let l = [(1, "a"); (2, "b"); (3, "c")]
let f = firsts l
let s = seconds l

let l = [(1, 2); (3, 4); (5, 6)]
let f = sum_pair_list l
let s = sum_pair_list2 l ;;
 val l : (int * int) list = [(1, 2); (3, 4); (5, 6)]
 val f : int = 21
 val s : int = 21
```