OUR FIRST* COMPLEX DATA STRUCTURE!
THE TUPLE

* it is really our second complex data structure since functions are data structures too!
A tuple is a fixed, finite, ordered collection of values
Some examples with their types:

(1, 2) : int * int

("hello", 7 + 3, true) : string * int * bool

('a', ("hello", "goodbye")) : char * (string * string)
To use a tuple, we extract its components

General case:

let (id1, id2, ..., idn) = e1 in e2

An example:

let (x,y) = (2,4) in x + x + y
To use a tuple, we extract its components

General case:

\[
\text{let } (id1, id2, \ldots, idn) = e1 \text{ in } e2
\]

An example:

\[
\text{let } (x,y) = (2,4) \text{ in } x + x + y \\
\rightarrow 2 + 2 + 4
\]
To use a tuple, we extract its components

General case:

\[
\text{let (id1, id2, \ldots, idn) = e1 in e2}
\]

An example:

\[
\text{let (x,y) = (2,4) in x + x + y}
\]

\[
\begin{align*}
\text{--> } & 2 + 2 + 4 \\
\text{--> } & 8
\end{align*}
\]
Rules for Typing Tuples

if e1 : t1 and e2 : t2
then (e1, e2) : t1 * t2
Rules for Typing Tuples

If $e_1 : t_1$ and $e_2 : t_2$ then $(e_1, e_2) : t_1 \times t_2$

If $e_1 : t_1 \times t_2$ then $x_1 : t_1$ and $x_2 : t_2$ inside the expression $e_2$

Let $(x_1, x_2) = e_1$ in $e_2$

Overall expression takes on the type of $e_2$
Problem:
• A point is represented as a pair of floating point values.
• Write a function that takes in two points as arguments and returns the distance between them as a floating point number.
Steps to writing functions over typed data:

1. Write down the function and argument names
2. Write down argument and result types
3. Write down some examples (in a comment)
Steps to writing functions over typed data:

1. Write down the function and argument names
2. Write down argument and result types
3. Write down some examples (in a comment)
4. Deconstruct input data structures
   • *the argument types suggests how to do it*
5. Build new output values
   • *the result type suggests how you do it*
Steps to writing functions over typed data:

1. Write down the function and argument names
2. Write down argument and result types
3. Write down some examples (in a comment)
4. Deconstruct input data structures
   • the argument types suggests how to do it
5. Build new output values
   • the result type suggests how you do it
6. Clean up by identifying repeated patterns
   • define and reuse helper functions
   • your code should be elegant and easy to read
Steps to writing functions over typed data:

1. Write down the function and argument names
2. Write down argument and result types
3. Write down some examples (in a comment)
4. Deconstruct input data structures
   • the argument types suggests how to do it
5. Build new output values
   • the result type suggests how you do it
6. Clean up by identifying repeated patterns
   • define and reuse helper functions
   • your code should be elegant and easy to read

Types help structure your thinking about how to write programs.
Distance between two points

a type abbreviation

type point = float * float
type point = float * float

let distance (p1 : point) (p2 : point) : float =

write down function name
argument names and types
Distance between two points

Let's define a point as a tuple of two floats:

```plaintext
type point = float * float
```

We can calculate the distance between two points using the Pythagorean theorem:

- Distance between (0.0, 0.0) and (0.0, 1.0) is 1.0
- Distance between (0.0, 0.0) and (1.0, 1.0) is \(\sqrt{1.0 + 1.0}\)

From the picture:
- Distance between \((x_1, y_1)\) and \((x_2, y_2)\) is \(\sqrt{a^2 + b^2}\)

Here's a function to calculate the distance between two points:

```plaintext
let distance (p1:point) (p2:point) : float =
```

Examples:

- Distance \((0.0, 0.0)\) and \((0.0, 1.0)\) is 1.0
- Distance \((0.0, 0.0)\) and \((1.0, 1.0)\) is \(\sqrt{1.0 + 1.0}\)
- From the picture:
  - Distance \((x_1, y_1)\) and \((x_2, y_2)\) is \(\sqrt{a^2 + b^2}\)
type point = float * float

let distance (p1: point) (p2: point) : float =

let (x1, y1) = p1 in
let (x2, y2) = p2 in
...

deconstruct function inputs
type point = float * float

let distance (p1:point) (p2:point) : float =
let (x1,y1) = p1 in
let (x2,y2) = p2 in

sqrt ((x2 -. x1) *. (x2 -. x1) +. (y2 -. y1) *. (y2 -. y1))

compute function results

notice operators on floats have a "." in them
type point = float * float

let distance (p1:point) (p2:point) : float =
    let square x = x *. x in
    let (x1,y1) = p1 in
    let (x2,y2) = p2 in
    sqrt (square (x2 -. x1)) +.
    square (y2 -. y1))
type point = float * float

let distance (p1:point) (p2:point) : float =
  let square x = x *. x in
  let (x1,y1) = p1 in
  let (x2,y2) = p2 in
  sqrt (square (x2 -. x1) +. square (y2 -. y1))

let pt1 = (2.0,3.0)
let pt2 = (0.0,1.0)
let dist12 = distance pt1 pt2
MORE TUPLES
Tuples

- Here's a tuple with 2 fields:

\[(4.0, 5.0) : \text{float} \times \text{float}\]
Tuples

• Here's a tuple with 2 fields:

   (4.0, 5.0) : float * float

• Here's a tuple with 3 fields:

   (4.0, 5, "hello") : float * int * string
Tuples

- Here's a tuple with 2 fields:
  
  \[(4.0, 5.0) : \text{float} \times \text{float}\]

- Here's a tuple with 3 fields:
  
  \[(4.0, 5, "hello") : \text{float} \times \text{int} \times \text{string}\]

- Here's a tuple with 4 fields:
  
  \[(4.0, 5, "hello", 55) : \text{float} \times \text{int} \times \text{string} \times \text{int}\]
Tuples

- Here's a tuple with 2 fields:
  
  \[(4.0, 5.0) : \text{float} \times \text{float}\]

- Here's a tuple with 3 fields:
  
  \[(4.0, 5, "hello") : \text{float} \times \text{int} \times \text{string}\]

- Here's a tuple with 4 fields:
  
  \[(4.0, 5, "hello", 55) : \text{float} \times \text{int} \times \text{string} \times \text{int}\]

- Have you ever thought about what a tuple with 0 fields might look like?
Unit

- **Unit** is the tuple with zero fields!

  ```
  () : unit
  ```

- the unit value is written with a pair of parens
- there are no other values with this type!
Unit

- **Unit** is the tuple with zero fields!

  `( ) : unit`

  - the unit value is written with an pair of parens
  - there are no other values with this type!

- Why is the unit type and value useful?
- Every expression has a type:

  `(print_string "hello world\n") : ???`
Unit

- **Unit** is the tuple with zero fields!

  \[
  () : \text{unit}
  \]

  - the unit value is written with an pair of parens
  - there are no other values with this type!

- Why is the unit type and value useful?
- Every expression has a type:

  \[
  (\text{print_string } "\text{hello world}\n") : \text{unit}
  \]

- Expressions executed for their *effect* return the unit value
OUR THIRD DATA STRUCTURE!
THE OPTION
Options

A value $v$ has type $t \text{ option}$ if it is either:

- the value $\text{None}$, or
- a value $\text{Some } v'$, and $v'$ has type $t$

Options can signal there is no useful result to the computation

Example: we look up a value in a hash table using a key.

- If the key is present, return $\text{Some } v$ where $v$ is the associated value
- If the key is not present, we return $\text{None}$
type point = float * float

let slope (p1:point) (p2:point) : float =
Slope between two points

**Type**: point = float * float

**Function**: let slope (p1:point) (p2:point) : float =

```
let (x1,y1) = p1 in
let (x2,y2) = p2 in
```

- Deconstruct tuple
Slope between two points

```ml
type point = float * float

let slope (p1:point) (p2:point) : float =
  let (x1,y1) = p1 in
  let (x2,y2) = p2 in
  let xd = x2 -. x1 in
  if xd != 0.0 then
    (y2 -. y1) /. xd
  else
    ???
```

What can we return?

Avoid divide by zero
type point = float * float

let slope (p1:point) (p2:point) : float option =
  let (x1,y1) = p1 in
  let (x2,y2) = p2 in
  let xd = x2 -. x1 in
  if xd != 0.0 then
    ???
  else
    ???

we need an option type as the result type
Slope between two points

**Type definitions:**
```haskell
type point = float * float

let slope (p1:point) (p2:point) : float option =
    let (x1,y1) = p1 in let (x2,y2) = p2 in
    let xd = x2 -. x1 in
    if xd != 0.0 then
        Some ((y2 -. y1) /. xd)
    else
        None
```
Slope between two points

type point = float * float

let slope (p1:point) (p2:point) : float option =
let (x1,y1) = p1 in let (x2,y2) = p2 in let xd = x2 -. x1 in
if xd != 0.0 then
  (y2 -. y1) /. xd
else
  None

(x1, y1)

(x2, y2)

Has type float

Can have type float option
Slope between two points

\[
\frac{y_2 - y_1}{x_2 - x_1}
\]

\[
\text{If } x_2 - x_1 \neq 0.0, \quad \frac{y_2 - y_1}{x_2 - x_1}
\]

\[
\text{Otherwise, } None
\]

**Type points**

\[
\text{type point} = \text{float} \times \text{float}
\]

**Function slope**

\[
\text{let slope (p1:point) (p2:point) : float option =}
\]

\[
\text{let (x1,y1) = p1 in}
\]

\[
\text{let (x2,y2) = p2 in}
\]

\[
\text{let xd = x2 -. x1 in}
\]

\[
\text{if xd != 0.0 then}
\]

\[
(y2 -. y1) /. xd
\]

\[
\text{else None}
\]

**Results**

\[
\text{Has type float}
\]

**Options**

\[
\text{Can have type float option}
\]

**Wrong**

\[
\text{WRONG: Type mismatch}
\]
type point = float * float

let slope (p1:point) (p2:point) : float option =
  let (x1,y1) = p1 in
  let (x2,y2) = p2 in
  let xd = x2 -. x1 in
  if xd != 0.0 then
    (y2 -. y1) /. xd
  else
    None

Has type float
doubly WRONG: result does not match declared result
Remember the typing rule for if

if e₁ : bool
and e₂ : t and e₃ : t (for some type t)
then if e₁ then e₂ else e₃ : t

Returning an optional value from an if statement:

if ... then

None : t option

else

Some ( ...) : t option
How do we use an option?

\[ \text{slope : point} \rightarrow \text{point} \rightarrow \text{float option} \]

returns a float option
How do we use an option?

```ocaml
slope : point -> point -> float option

let print_slope (p1:point) (p2:point) : unit =
```
How do we use an option?

\[
\text{slope} : \text{point} \rightarrow \text{point} \rightarrow \text{float option}
\]

let \text{print_slope} (p1:\text{point}) (p2:\text{point}) : \text{unit} = \\
\text{slope p1 p2}

returns a float option; to print we must discover if it is None or Some
How do we use an option?

slope : point -> point -> float option

let print_slope (p1:point) (p2:point) : unit =
  match slope p1 p2 with
How do we use an option?

slope : point -> point -> float option

let print_slope (p1:point) (p2:point) : unit =
  match slope p1 p2 with
  Some s ->
  | None ->

There are two possibilities

Vertical bar separates possibilities
How do we use an option?

`slope : point -> point -> float option`

```ocaml
let print_slope (p1:point) (p2:point) : unit =
    match slope p1 p2 with
    | Some s ->
    | None ->
```

The object between | and -> is called a pattern

The "Some s" pattern includes the variable s
How do we use an option?

`slope : point -> point -> float option`

```ocaml
let print_slope (p1:point) (p2:point) : unit =
  match slope p1 p2 with
  | Some s ->
  | None ->
```

You can put a “|” on the first line if you want. It is generally considered better style to do so. When I learned OCaml, that wasn’t an option so I forget to do it a lot...
How do we use an option?

slope : point -> point -> float option

let print_slope (p1:point) (p2:point) : unit =
  match slope p1 p2 with
  Some s ->
    print_string ("Slope: " ^ string_of_float s)
  | None ->
    print_string "Vertical line.\n"
Steps to writing functions over typed data:

1. Write down the function and argument names
2. Write down argument and result types
3. Write down some examples (in a comment)
4. **Deconstruct** input data structures
5. **Build** new output values
6. Clean up by identifying repeated patterns

For option types:

- when the **input** has type `t option`, deconstruct with:
  
  ```
  match ... with
  | None -> ...
  | Some s -> ...
  ```

- when the **output** has type `t option`, construct with:
  
  ```
  Some (...)    None
  ```
MORE PATTERN MATCHING
Recall the Distance Function

```ocaml
type point = float * float

let distance (p1:point) (p2:point) : float =
    let square x = x *. x in
    let (x1,y1) = p1 in
    let (x2,y2) = p2 in
    sqrt (square (x2 -. x1) +. square (y2 -. y1))
```
Recall the Distance Function

```ocaml
type point = float * float

let distance (p1:point) (p2:point) : float =
    let square x = x *. x in
    let (x1,y1) = p1 in
    let (x2,y2) = p2 in
    sqrt (square (x2 -. x1) +. square (y2 -. y1))
```

(x2, y2) is an example of a pattern – a pattern for tuples.

So let declarations can contain patterns just like match statements

The difference is that a match allows you to consider multiple different data shapes
Recall the Distance Function

```ocaml
type point = float * float

let distance (p1:point) (p2:point) : float =
  let square x = x *. x in
  match p1 with
  | (x1,y1) ->
    let (x2,y2) = p2 in
    sqrt (square (x2 -. x1) +. square (y2 -. y1))
```

There is only 1 possibility when matching a pair
Recall the Distance Function

We can nest one match expression inside another. (We can nest any expression inside any other, if the expressions have the right types)
Better Style: Complex Patterns

we built a pair of pairs

```ocaml
type point = float * float

let distance (p1:point) (p2:point) : float =
    let square x = x *. x in
    match (p1, p2) with
    | ((x1,y1), (x2, y2)) ->
      sqrt (square (x2 -. x1) +. square (y2 -. y1))
```

Pattern for a pair of pairs: ((variable, variable), (variable, variable))
All the variable names in the pattern must be different.
type point = float * float

let distance (p1:point) (p2:point) : float =
  let square x = x *. x in
  match (p1, p2) with
  | (p3, p4) ->
    let (x1, y1) = p3 in
    let (x2, y2) = p4 in
    sqrt (square (x2 -. x1) +. square (y2 -. y1))

A pattern must be **consistent with** the type of the expression in between `match ... with`
We use `(p3, p4)` here instead of `((x1, y1), (x2, y2))`
Pattern-matching in function parameters

```
type point = float * float

let distance ((x1,y1):point) ((x2,y2):point) : float =
    let square x = x *. x in
    sqrt (square (x2 -. x1) +. square (y2 -. y1))
```

Function parameters are patterns too!
What’s the best style?

Either of these is reasonably clear and compact.
Code with unnecessary nested matches/lets is particularly ugly to read.
You'll be judged on code style in this class.
What’s the best style?

let distance (x1,y1) (x2,y2) =
  let square x = x *. x in
  sqrt (square (x2 -. x1) +. square (y2 -. y1))

This is how I'd do it ... the types for tuples + the tuple patterns are a little ugly/verbose ... but for now in class, use the explicit type annotations. We will loosen things up later in the semester.
type point = float * float

(* returns a nearby point in the graph if one exists *)
nearby : graph -> point -> point option

let printer (g:graph) (p:point) : unit =
  match nearby g p with
  | None -> print_string "could not find one\n"
  | Some (x,y) ->
    print_float x;
    print_string ", ";
    print_float y;
    print_newline();
Other Patterns

Constant values can be used as patterns

```ocaml
let small_prime (n:int) : bool =
  match n with
  | 2 -> true
  | 3 -> true
  | 5 -> true
  | _ -> false
```

```ocaml
let iffy (b:bool) : int =
  match b with
  | true -> 0
  | false -> 1
```

the underscore pattern
matches anything
it is the "don't care" pattern
SUMMARY:
BASIC FUNCTIONAL PROGRAMMING
Steps to writing functions over typed data:

1. Write down the function and argument names
2. Write down argument and result types
3. Write down some examples (in a comment)
4. **Deconstruct** input data structures
5. **Build** new output values
6. Clean up by identifying repeated patterns

For tuple types:

- when the **input** has type \( t_1 \times t_2 \)
  - use `let (x, y) = ...` to deconstruct
- when the **output** has type \( t_1 \times t_2 \)
  - use `(e1, e2)` to construct

We will see this paradigm repeat itself over and over
Reading Assignments

• Lecture Notes 03: Type-directed Programming

• Optional: Book “Real World OCaml”
  • Chapter 2
WHERE DID TYPE SYSTEMS COME FROM?
Origins of Type Theory

Georg Cantor

Origins of Type Theory

Über eine Eigenschaft des Inbegriffes aller reellen algebraischen Zahlen. 1874

(On a Property of the System of all the Real Algebraic Numbers)

“Considered the first purely theoretical paper on set theory.” *

Georg Cantor

Origins of Type Theory

Bertrand Russell
He noticed that Cantor’s set theory allows the definition of this set $S$:

$$\{ A \mid A \text{ is a set and } A \notin A \}$$

Bertrand Russell
He noticed that Cantor’s set theory allows the definition of this set S:

\[ \{ A \mid A \text{ is a set and } A \notin A \} \]

If we assume S is not in the set S, then by definition, it must belong to that set.

If we assume S is in the set S, then it contradicts the definition of S.

Russell’s paradox
He noticed that Cantor’s set theory allows the definition of this set $S$:

$$\{ A \mid A \text{ is a set and } A \notin A \}$$

Russell’s solution:

Each set has a distinct type:

type 1, 2, 3, 4, 5, ...

A set of type $i+1$ can only have elements of type $i$ so it can’t include itself.
Developers of Zermelo-Fraenkel set theory (1921).
An alternative solution to Russell’s paradox.
Fast Forward to the 70s

Robin Milner

In 1978, developed ML and coined the phrase “well-typed programs don’t go wrong” to describe a key property of type-safe languages.
Well-typed Programs Don’t Go Wrong

Some ML programs do not have a well-defined semantics:

“hello” + 3

Such programs do not type check.
Some ML programs do not have a well-defined semantics:

```
let x = "hello" in
let y = 3 in
x + y
```

"hello" + 3

Such programs do not type check.

Moreover, when we execute a well-typed program, we are guaranteed to never, ever run into such a program during execution.
Some ML programs do not have a well-defined semantics:

```
let x = "hello" in
let y = 3 in
x + y
```

"hello" + 3

Such programs do not type check.

Moreover, when we execute a well-typed program, we are guaranteed to never, ever run into such a program during execution.

```
let x = "hello" in
let y = 3 in
x + y
```

-->*

"hello" + 3

well-typed programs don’t reduce to programs like "hello" + 3, which go wrong
Well-type programs don’t go wrong

What about this expression:

3 / 0
Well-type programs don’t go wrong

What about this expression:

\[
\frac{3}{0}
\]

It type checks. When executed, ML will supply this message:

```
Exception: Division_by_zero.
```

Did the expression “go wrong”? Did it violate our credo “well-typed expressions don’t go wrong?”
Well-type programs don’t go wrong

What about this expression:

\[ \frac{3}{0} \]

It type checks. When executed, ML will supply this message:

Exception: Division_by_zero.

Did the expression “go wrong”? Did it violate our credo “well-typed expressions don’t go wrong?”

No and No. Exceptions are a well-defined result of a computation. ie: you can look up what happens to 3 / 0 in the OCaml manual.
What’s the difference between raising an exception and “going wrong”?

Why distinguish between these things?

Does one have to treat “hello” + 3 as “going wrong”?

Why does OCaml make such choices?

Is it reasonable for other languages to choose differently?
Type Soundness

“*Well typed programs do not go wrong*”

Programming languages with this property have *sound* type systems. They are called *safe* languages.

Safe languages are generally *immune* to buffer overrun vulnerabilities, uninitialized pointer vulnerabilities, etc., etc. (but not immune to all bugs!)

Safe languages: ML, Java, Python, ...

Unsafe languages: C, C++, Pascal
“Well typed programs do not go wrong”
Robin Milner, 1978

Robin Milner

Turing Award, 1991

“For three distinct and complete achievements:

1. LCF, the mechanization of Scott's Logic of Computable Functions, probably the first theoretically based yet practical tool for machine assisted proof construction;

2. ML, the first language to include polymorphic type inference together with a type-safe exception-handling mechanism;

3. CCS, a general theory of concurrency.

In addition, he formulated and strongly advanced full abstraction, the study of the relationship between operational and denotational semantics.”