Simple Functional Data

COS 326
Andrew Appel
Princeton University

What is the single most important mathematical concept ever developed in human history?

What is the single most important mathematical concept ever developed in human history?

An answer: The mathematical variable

What is the single most important mathematical concept ever developed in human history?

An answer: The mathematical variable

(runner up: natural numbers/induction)

Why is the mathematical variable so important?

The mathematician says:

“Let x be some integer, we define a polynomial over x ...”

Why is the mathematical variable so important?

The mathematician says:

"Let x be some integer, we define a polynomial over x ..."

What is going on here? The mathematician has separated a *definition* (of x) from its *use* (in the polynomial).

This is the most primitive kind of *abstraction* (x is *some* integer)

Abstraction is the key to controlling complexity and without it, modern mathematics, science, and computation would not exist.

It allows *reuse* of ideas, theorems ... functions and programs!

OCAML BASICS:
LET DECLARATIONS

Abstraction & Abbreviation

In OCaml, the most basic technique for factoring your code is to use let expressions

Instead of writing this expression:

```
(2 + 3) * (2 + 3)
```

We write this one:

```
let x = 2 + 3 in
x * x
```

A Few More Let Expressions

```
let x = 2 in
let squared = x * x in
let cubed = x * squared in
squared * cubed
```

```
let a = "a" in
let b = "b" in
let as = a ^ a ^ a in
let bs = b ^ b ^ b in
as ^ bs
```

A Technical Note: The Structure of a .ml File

Foo.ml

```
<declaration>

<declaration>

      …
```

Every .ml file is a sequence of *declarations*

These “declarations” are a little different than “expressions”

A Technical Note: The Structure of a .ml File

Bar.ml

```
let x = 17 + 5

let y = x + 22
```

Bar.ml contains two *let declarations*

Let declarations do not end with “in”

Let declarations have the form:

let *<var>* = *<expression>*

A Technical Note: The Structure of a .ml File

Baz.ml

```
let x =
  let z = 22 in
  z + z

let y =
  if x < 17 then
    let w = x + 1 in
    2 * w
  else
    26
```

Because let declarations have this form:

let *<var>* = *<expression>*

they contain expressions

... including “let expressions” which have the form:

let *<var>* = *<expression>* in *<expression>*

OCaml Variables are Immutable

Once *bound* to a value, a variable is never modified or changed.

```
let x = 3

let add_three (y:int) : int = y + x
```

given a *use* of a variable, like this one for *x*, work outwards and upwards through a program to find the closest enclosing *definition*. That is the value of this use *forever and always*.

OCaml Variables are Immutable

Once *bound* to a value, a variable is never modified or changed.

```
let x = 3

let add_three (y:int) : int = y + x
```

given a *use* of a variable, like this one for *x*, work outwards and upwards through a program to find the closest enclosing *definition*. That is the value of this use *forever and always*.

OCaml Variables are Immutable

Once *bound* to a value, a variable is never modified or changed.

```
let x = 3

let add_three (y:int) : int = y + x
```

given a *use* of a variable, like this one for *x*, work outwards and upwards through a program to find the closest enclosing *definition*. That is the value of this use *forever and always*.

OCaml Variables are Immutable

Once *bound* to a value, a variable is never modified or changed.

```
let x = 3

let add_three (y:int) : int = y + x
```

It does not matter what I write next. add_three will always add 3!

OCaml Variables are Immutable

Once *bound* to a value, a variable is never modified or changed.

a distinct variable that "happens to be spelled the same"

```
let x = 3

let add_three (y:int) : int = y + x

let x = 4

let add_four (y:int) : int = y + x
```

OCaml Variables are Immutable

A use of a variable always refers to it's *closest* (in terms of syntactic distance) enclosing declaration. Hence, we say OCaml is a *statically scoped* (or *lexically scoped*) language

we can use add_three without worrying about the second definition of x

```
let x = 3

let add_three (y:int) : int = y + x

let x = 4

let add_four (y:int) : int = y + x

let add_seven (y:int) : int =
  add_three (add_four y)
```

OCaml Variables are Immutable

Since the two variables (both happened to be named x) are actually different, unconnected things, we can rename them.

This is known as *alpha-conversion*.

you can rename
x to zzz
by replacing
the definition
and all its uses with
the new name

```
let x = 3

let add_three (y:int) : int = y + x

let x = 4

let add_four (y:int) : int = y + x

let add_seven (y:int) : int =
  add_three (add_four y)
```

OCaml Variables are Immutable

Since the two variables (both happened to be named x) are actually different, unconnected things, we can rename them.

This is known as *alpha-conversion*.

you can rename
x to zzz
by replacing
the definition
and all its uses with
the new name

```
let x = 3

let add_three (y:int) : int = y + x

let zzz = 4

let add_four (y:int) : int = y + zzz

let add_seven (y:int) : int =
  add_three (add_four y)
```

How does OCaml execute a let expression?

```
let x = <expression1> in
<expression2>
```

In a nutshell:

- execute <expression1>, until you get a value v1
- substitute that value v1 for x in <expression2>
- execute <expression2>, until you get a value v2
- the result of the whole execution is v2

How does OCaml execute a let expression?

```
let x = 2 + 1 in x * x
```

-->

```
let x = 3 in x * x
```

How does OCaml execute a let expression?

```
let x = 2 + 1 in x * x
```

-->

```
let x = 3 in x * x
```

substitute
3 for x

-->

```
        3 * 3
```

How does OCaml execute a let expression?

```
let x = 2 + 1 in x * x
```

-->

```
let x = 3 in x * x
```

substitute
3 for x

-->

```
3 * 3
```

-->

```
9
```

How does OCaml execute a let expression?

```
let x = 2 + 1 in x * x
```

-->

```
let x = 3 in x * x
```

substitute
3 for x

-->

```
3 * 3
```

-->

```
9
```

Note: I write
e1 --> e2
when e1 evaluates
to e2 in one step

Meta-comment

OCaml expression

OCaml expression

```
let x = 2 in x + 3      -->      2 + 3
```

I defined the language in terms of itself:
By reduction of one OCaml expression to another

I'm trying to train you to think at a high level of abstraction.

I didn't have to mention low-level abstractions like assembly code or registers or memory layout to tell you how OCaml works.

Another Example

```
let x = 2 in
let y = x + x in
y * x
```

Another Example

```
let x = 2 in
let y = x + x in
y * x
```

substitute
2 for x

-->

```
let y = 2 + 2 in
y * 2
```

Another Example

```
let x = 2 in
let y = x + x in
y * x
```

substitute
2 for x

-->

```
let y = 2 + 2 in
y * 2
```

-->

```
let y = 4      in
y * 2
```

Another Example

```
let x = 2 in
let y = x + x in
y * x
```

substitute
2 for x

-->

```
let y = 2 + 2 in
y * 2
```

-->

```
let y = 4       in
y * 2
```

substitute
4 for y

-->

```
4 * 2
```

Another Example

```
let x = 2 in
let y = x + x in
y * x
```

substitute
2 for x

-->

```
let y = 2 + 2 in
y * 2
```

-->

```
let y = 4       in
y * 2
```

substitute
4 for y

-->

```
4 * 2
```

-->

```
8
```

Moral: Let operates by *substituting* computed values for variables

OCAML BASICS: TYPE CHECKING AGAIN

Back to Let Expressions ... Typing

x granted type of e1 for use in e2

```
let x = e1 in

e2
```

overall expression
takes on the type of e2

Back to Let Expressions ... Typing

x granted type of e1 for use in e2

```
let x = e1 in

e2
```

overall expression
takes on the type of e2

x has type int
for use inside the
let body

```
let x = 3 + 4 in

string_of_int x
```

overall expression
has type string

Let Expressions Really Are Expressions

$$\boxed{2 + 3}$$

an expression

Let Expressions Really Are Expressions

```
2 + 3
```

an expression

```
let x = 2 + 3 in
x + x
```

an expression

Let Expressions Really Are Expressions

```
2 + 3
```

an expression

```
let x = 2 + 3 in
x + x
```

an expression

an expression

```
let x = let y = 2 + 3 in y + 5 in
1 + x
```

let expressions can appear anywhere other expressions can appear. they can be *nested*

Exercise

(a)

```
let x =
  let y = 2 + 3 in y
in
  let x = "1" in
x + x
```

(b)

```
let x =
  let y = "2" ^ "3" in y
in
  let x = 1 in
x + x
```

Which of (a) or (b) type check? Explain why.

On a piece of paper (or in your favorite editor), show the step-by-step evaluation of the example that type checks.

Critique the *programming style* used in these examples.

OCAML BASICS: FUNCTIONS

Defining functions

```
let add_one (x:int) : int = 1 + x
```

Defining functions

let keyword

```
let add_one (x:int) : int = 1 + x
```

function name

argument name

type of argument

type of result

expression that computes value produced by function

Note: recursive functions with begin with "**let rec**"

Defining functions

Nonrecursive functions:

```
let add_one (x:int) : int = 1 + x

let add_two (x:int) : int = add_one (add_one x)
```

definition of add_one
must come before use

Defining functions

Nonrecursive functions:

```
let add_one (x:int) : int = 1 + x

let add_two (x:int) : int = add_one (add_one x)
```

With a local definition:

local function definition hidden from clients

```
let add_two' (x:int) : int =
  let add_one x = 1 + x in
  add_one (add_one x)
```

I left off the types. OCaml figures them out

Good style: types on top-level definitions

Types for Functions

Some functions:

```
let add_one (x:int) : int = 1 + x

let add_two (x:int) : int = add_one (add_one x)

let add (x:int) (y:int) : int = x + y
```

function with two arguments

Types for functions:

```
add_one : int -> int

add_two : int -> int

add : int -> int -> int
```

Rule for type-checking functions

General Rule:

If a function f : T1 → T2
and an argument e : T1
then f e : T2

Example:

```
add_one : int -> int

3 + 4 : int

add_one (3 + 4) : int
```

Rule for type-checking functions

Recall the type of add:

Definition:

```
let add (x:int) (y:int) : int =
  x + y
```

Type:

```
add : int -> int -> int
```

Rule for type-checking functions

Recall the type of add:

Definition:

```
let add (x:int) (y:int) : int =
  x + y
```

Type:

```
add : int -> int -> int
```

Same as:

```
add : int -> (int -> int)
```

Rule for type-checking functions

General Rule:

If a function $f : T1 \rightarrow T2$
and an argument $e : T1$
then $f\ e : T2$

$A \rightarrow B \rightarrow C$

same as:

$A \rightarrow (B \rightarrow C)$

Example:

```
add : int -> int -> int

3 + 4 : int

add (3 + 4) : ???
```

Rule for type-checking functions

General Rule:

If a function f : T1 -> T2
and an argument e : T1
then f e : T2

A -> B -> C

same as:

A -> (B -> C)

Example:

```
add : int -> (int -> int)

3 + 4 : int

add (3 + 4) :
```

Rule for type-checking functions

General Rule:

If a function f : T1 -> T2
and an argument e : T1
then f e : T2

A -> B -> C

same as:

A -> (B -> C)

Example:

```
add : int -> (int -> int)

3 + 4 : int

add (3 + 4) : int -> int
```

Rule for type-checking functions

General Rule:

If a function f : T1 -> T2
and an argument e : T1
then f e : T2

A -> B -> C

same as:

A -> (B -> C)

Example:

```
add : int -> int -> int

3 + 4 : int

add (3 + 4) : int -> int

(add (3 + 4)) 7 : int
```

Rule for type-checking functions

General Rule:

If a function f : T1 -> T2
and an argument e : T1
then f e : T2

A -> B -> C

same as:

A -> (B -> C)

Example:

```
add : int -> int -> int

3 + 4 : int

add (3 + 4) : int -> int

add (3 + 4) 7 : int
```

extra parens
not necessary

One key thing to remember

- If you have a function f with a type like this:

$$A \to B \to C \to D \to E \to F$$

- Then each time you add an argument, you can get the type of the result by knocking off the first type in the series

f a1 : $B \to C \to D \to E \to F$ (if a1 : A)

f a1 a2 : $C \to D \to E \to F$ (if a2 : B)

f a1 a2 a3 : $D \to E \to F$ (if a3 : C)

f a1 a2 a3 a4 a5 : F (if a4 : D and a5 : E)

TYPE ERRORS

Type Checking Rules

Type errors for if statements can be confusing sometimes. Recall:

```
let rec concatn s n =
  if n <= 0 then
    ...
  else
    s ^ (concatn s (n-1))
```

Type Checking Rules

Type errors for if statements can be confusing sometimes. Recall:

```
let rec concatn s n =
  if n <= 0 then
    ...
  else
    s ^ (concatn s (n-1))
```

ocaml might point to (concatn s (n-1)) and says:

```
Error: This expression has type int but an
expression was expected of type string
```

Type Checking Rules

Type errors for if statements can be confusing sometimes. Recall:

```
let rec concatn s n =
  if n <= 0 then
    ...
  else
    s ^ (concatn s (n-1))
```

ocaml might say:

```
Error: This expression has type int but an
expression was expected of type string
```

or ocaml might point to the expression (s ^ (concatn ...)) and say:

```
Error: This expression has type string but an
expression was expected of type int
```

Type Checking Rules

Type errors for if statements can be confusing sometimes.

Example. We create a string from s, concatenating it n times:

```
let rec concatn s n =
  if n <= 0 then
    ...
  else
    s ^ (concatn s (n-1))
```

???

```
Error: This expression has type int but an
expression was expected of type string
```

```
Error: This expression has type string but an
expression was expected of type int
```

Type Checking Rules

Type errors for if statements can be confusing sometimes.
Example. We create a string from s, concatenating it n times:

they don't *agree*!

```
let rec concatn s n =
  if n <= 0 then
    0
  else
    s ^ (concatn s (n-1))
```

???

```
Error: This expression has type int but an
expression was expected of type string
```

```
Error: This expression has type string but an
expression was expected of type int
```

Type Checking Rules

Type errors for if statements can be confusing sometimes.
Example. We create a string from s, concatenating it n times:

they don't *agree*!

```
let rec concatn s n =
  if n <= 0 then
    0
  else
    s ^ (concatn s (n-1))
```

???

The type checker points to *some* place where there is *disagreement*.

Moral: *Sometimes you need to look in an earlier branch for the error* even though the type checker points to a later branch.
The type checker doesn't know what the user wants.

A Tactic: Add Typing Annotations

```
let rec concatn (s:string) (n:int) : string =
  if n <= 0 then
    0
  else
    s ^ (concatn s (n-1))
```

```
Error: This expression has type int but an
expression was expected of type string
```

Exercise

Given the following code:

```
let munge b x =
  if not b then
    string_of_int x
  else
    "hello"

let y = 17
```

What are the types of the following expressions?
(And what must the types of f and g be?)

```
munge : ??

munge (y > 17) : ??

munge true (f (munge false 3)) : ??

munge true (g munge) : ??
```

DATA STRUCTURES: THE TUPLE

* it is really our second complex data structure since functions are data structures too!

Tuples

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)
```

Tuples

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
```

Tuples

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
-->  2 + 2 + 4
```

substitute!

Tuples

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
-->  2 + 2 + 4
-->  8
```

Rules for Typing Tuples

if e1 : t1 and e2 : t2
then (e1, e2) : t1 * t2

Rules for Typing Tuples

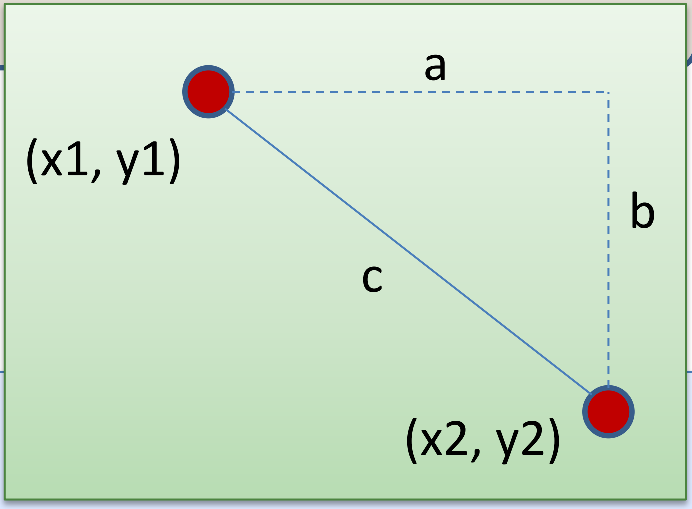
if e1 : t1 and e2 : t2
then (e1, e2) : t1 * t2

if e1 : t1 * t2 then
x1 : t1 and x2 : t2
inside the expression e2

```
let (x1,x2) = e1 in

e2
```

overall expression
takes on the type of e2

Distance between two points



$c^2 = a^2 + b^2$

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

Writing Functions Over Typed Data

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)

Writing Functions Over Typed Data

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*

Writing Functions Over Typed Data

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

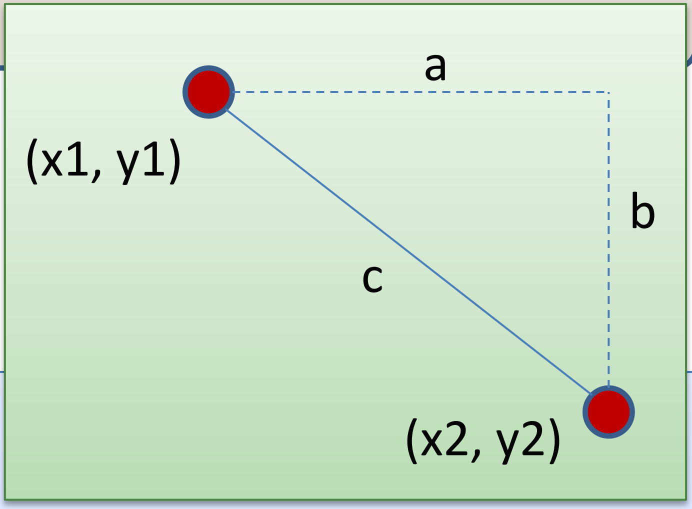
Writing Functions Over Typed Data

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.

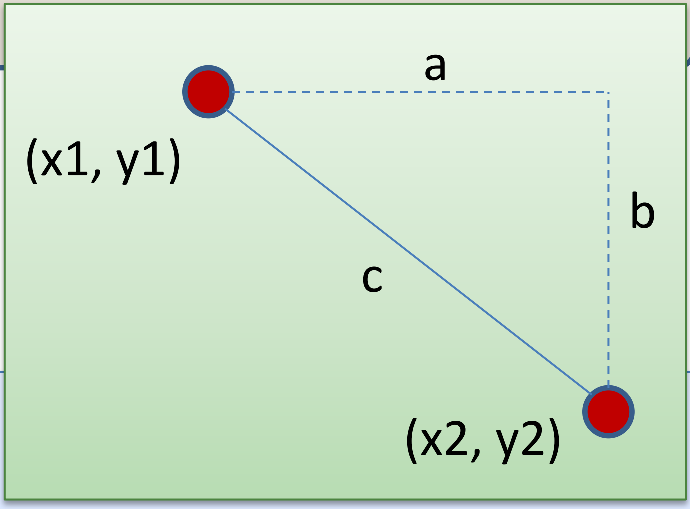
Distance between two points



a type abbreviation

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

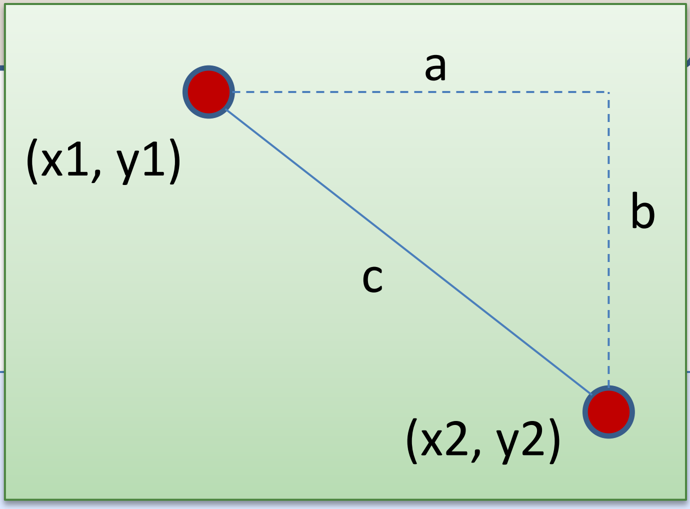
Distance between two points




```
type point = float * float

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

write down function name
argument names and types

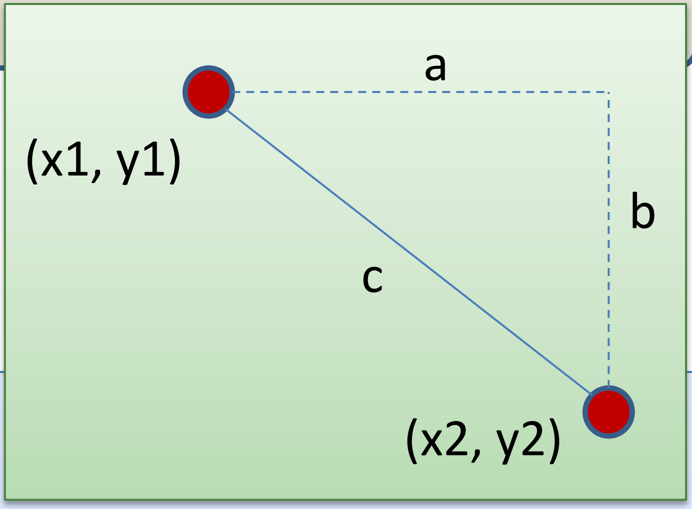
Distance between two points



examples

```
type point = float * float


(* distance (0.0,0.0) (0.0,1.0) == 1.0
 * distance (0.0,0.0) (1.0,1.0) == sqrt(1.0 + 1.0)
 *
 * from the picture:
 * distance (x1,y1) (x2,y2) == sqrt(a^2 + b^2)
 *)


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

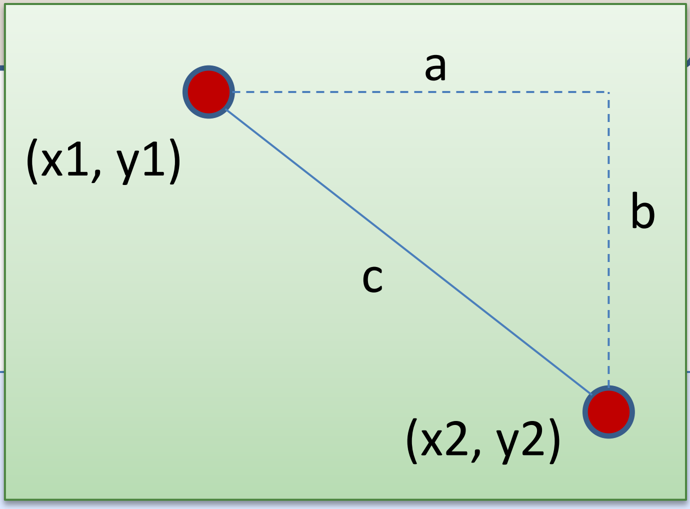
Distance between two points




```
type point = float * float

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

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

deconstruct
function inputs

Distance between two points

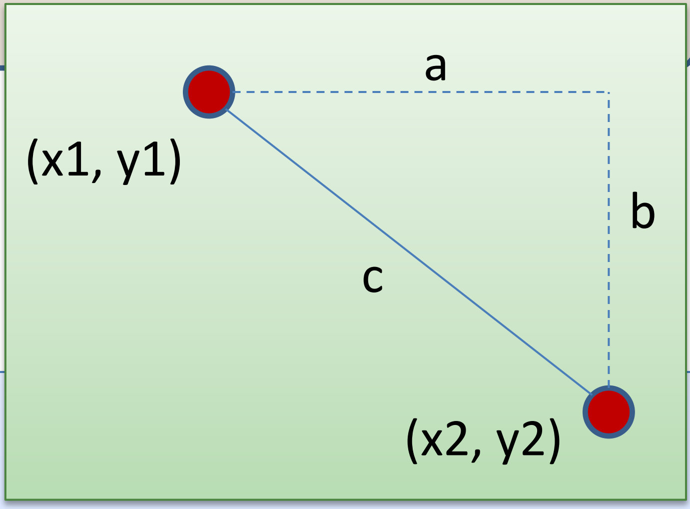



```
type point = float * float

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

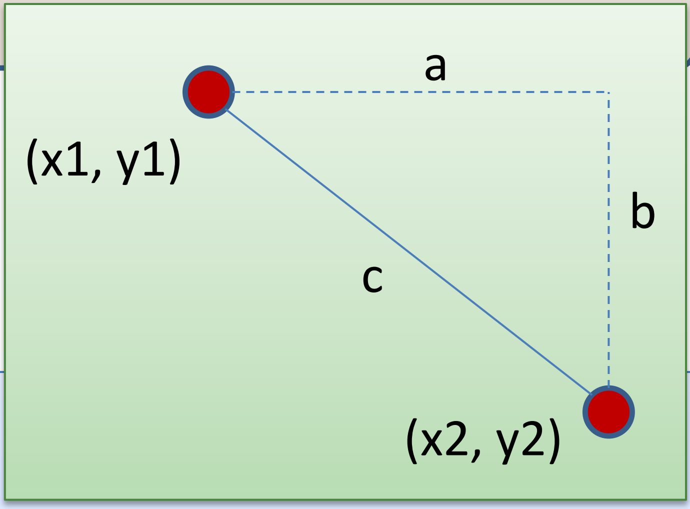
Distance between two points




```
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))
```

define helper functions to avoid repeated code

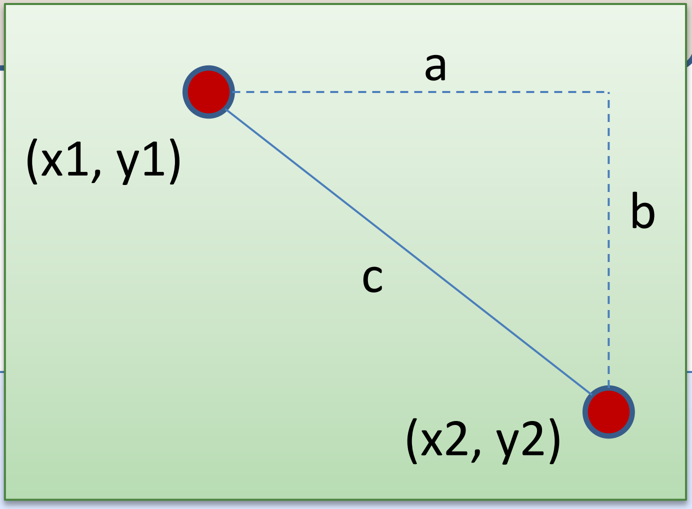
Distance between two points




```
type point = float * float

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

use tuple patterns
in function arguments
if you'd like

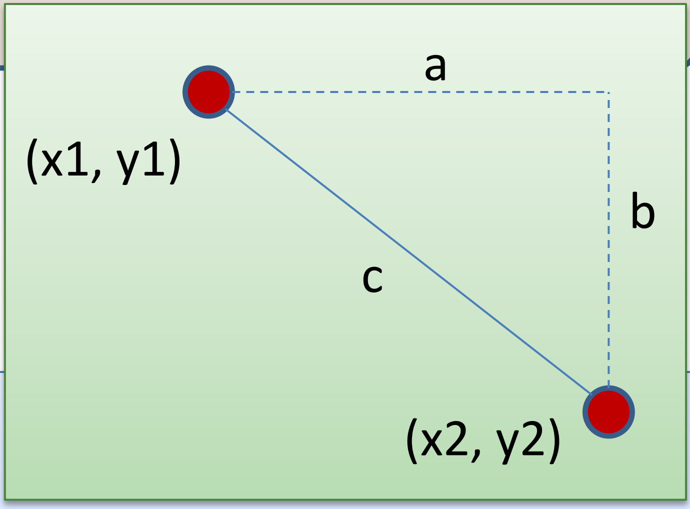
Distance between two points




```
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))
```

type annotations
can be included

Distance between two points




```
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
```

implement some tests

MORE TUPLES

Tuples

Here's a tuple with 2 fields:

(4.0, 5.0) : float * 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) : float * float

Here's a tuple with 3 fields:

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

Here's a tuple with 4 fields:

(4.0, 5, "hello", 55) : float * int * string * int

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

Here's a tuple with 4 fields:

(4.0, 5, "hello", 55) : float * int * string * int

Here's a tuple with 0 fields:

() : unit

SUMMARY:
BASIC FUNCTIONAL PROGRAMMING

Writing Functions Over Typed Data

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 t1 * t2
  - use let (x,y) = … to deconstruct
- when the output has type t1 * t2
  - use (e1, e2) to construct

We will see this paradigm repeat itself over and over

Records

Records are a lot like tuples. It's just that they have named fields.

Having named fields (records rather than tuples) often makes it easier to understand a program, especially when there are more than just 2 or 3 fields in a structure.

Records

Records are a lot like tuples. It's just that they have named fields.

Having named fields (records rather than tuples) often makes it easier to understand a program, especially when there are more than just 2 or 3 fields in a structure.

An example:

```
type name = {first:string; last:string;}

let my_name = {first="David"; last="Walker";}

let to_string (n:name) = n.last ^ ", " ^ n.first
```

Records

Records are a lot like tuples. It's just that they have named fields.

Having named fields (records rather than tuples) often makes it easier to understand a program, especially when there are more than just 2 or 3 fields in a structure.

An example:

```
type name = {first:string; last:string;}

let my_name = {first="David"; last="Walker";}

let to_string (n:name) = n.last ^ ", " ^ n.first
```

Note: Records come with several other useful features, like functional updates via "with expressions."

See *Real World OCaml* for more info.

WRAP-UP

Writing Functions Over Typed Data

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 t1 * t2
  - use let (x,y) = … to deconstruct
- when the output has type t1 * t2
  - use (e1, e2) to construct

We will see this paradigm repeat itself over and over

Exercise

What error do you get when you try to compile this file? (Type it in.) Why?

```
type item = {
   number: int;
   name: string;
}

type contact = {
   name: string*string;   (* first and last name *)
   phone: item;
}

let get_name x = x.name

let myphone = {number=122; name="iphone";}

let _ = print_endline (get_name myphone)
```