Simple Data

COS 326
David Walker
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*.

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

OCAML BASICS:
LET DECLARATIONS

Abstraction

- Good programmers identify repeated patterns in their code and factor out the repetition into meaningful components
- In O'Caml, the most basic technique for factoring your code is to use let expressions
- Instead of writing this expression:

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

Abstraction & Abbreviation

- Good programmers identify repeated patterns in their code and factor out the repetition into meaning components
- In O'Caml, 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
```

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
```

Abstraction & Abbreviation

- Two kinds of let:

```
if tuesday() then
     let x = 2 + 3 in
     x + x
else
     0
;;
```

```
let x = 2 + 3 ;;

let y = x + 17 / x ;;
```

let … **in** … is an *expression* that can appear inside any other *expression*

The scope of x does not extend outside the enclosing "in"

let … ;; without "in" is a top-level *declaration*

Variables x and y may be exported; used by other modules

(Don't need ;; if another let comes next; do need it if expression next)

Binding Variables to Values

- Each O'Caml variable is *bound* to 1 value
- *The value to which a variable is bound to never changes*!

```
let x = 3 ;;

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

Binding Variables to Values

- Each O'Caml variable is *bound* to 1 value
- *The value to which a variable is bound to never changes*!

```
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!

Binding Variables to Values

- Each O'Caml variable is bound to 1 value
- *The value a variable is bound to never changes*!

```
let x = 3 ;;

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

let x = 4 ;;

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

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

Binding Variables to Values

- Since the 2 variables (both happened to be named x) are actually different, unconnected things, we can rename them

rename x to zzz if you want to, replacing its uses

```
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)
;;
```

Binding Variables to Values

- Each O'Caml variable is bound to 1 value
- O'Caml is a statically 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)
;;
```

How do let expressions operate?

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

How do let expressions operate?

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

-->

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

How do let expressions operate?

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

-->

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

substitute
3 for x

-->

```
3 * 3
```

How do let expressions operate?

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

-->

let x = 3 in x * x

-->

        3 * 3

-->

          9
```

substitute
3 for x

How do let expressions operate?

```
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

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

Typing Simple Let Expressions

x granted type of e1 for use in e2

```
let x = e1 in

e2
```

overall expression
takes on the type of e2

Typing Simple Let Expressions

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
```

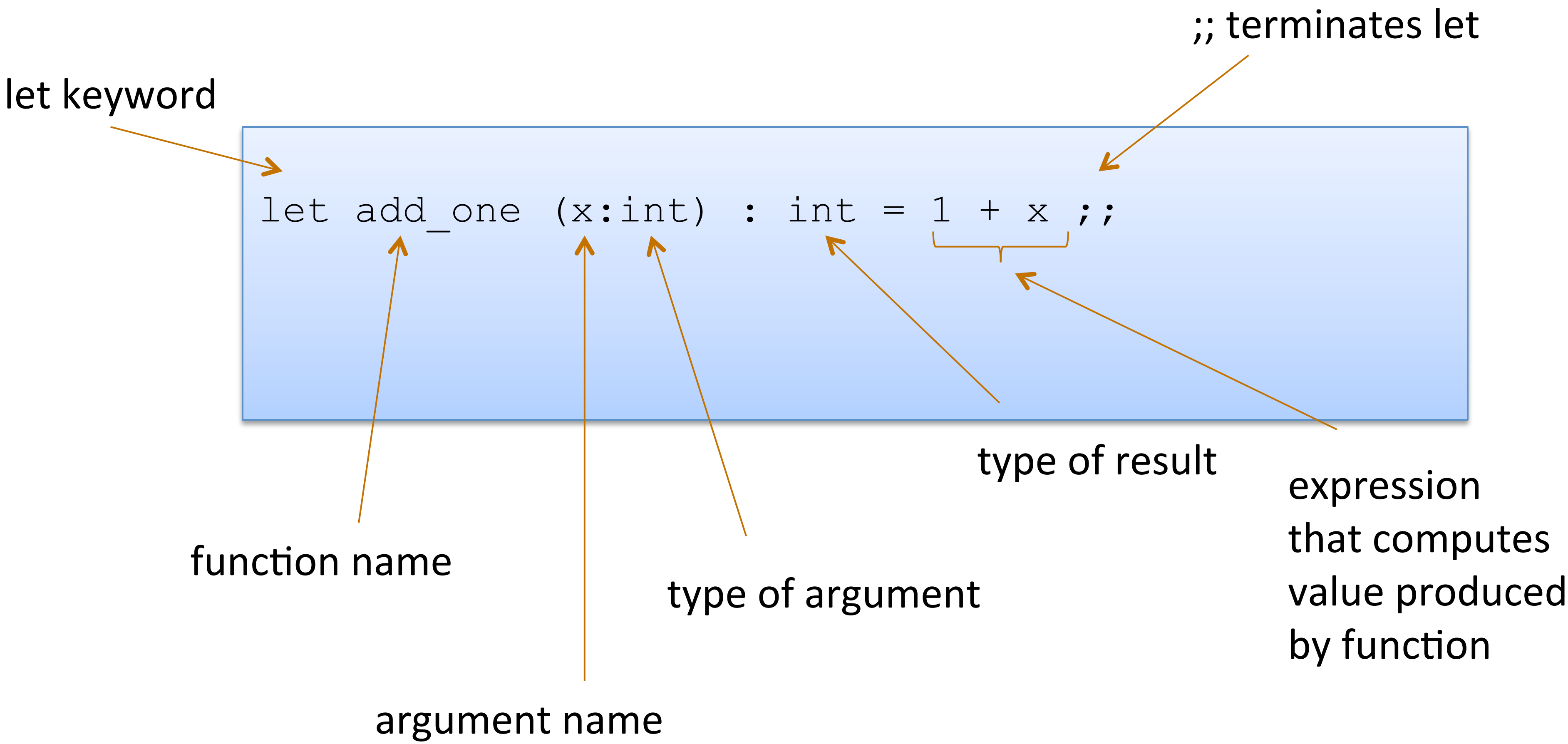
overall expression has type string

OCAML BASICS: FUNCTIONS

Defining functions

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

Defining functions



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

Defining functions

- Non-recursive 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

- Non-recursive 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. O'Caml 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 -> T2
and an argument e : T1
then f e : T2

Note:

A -> B -> C
is the 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

Remember:

A -> B -> C
is the 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

Remember:

A -> B -> C
is the 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

Remember:

A -> B -> C
is the 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

Remember:

A -> B -> C
is the 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

Example:

```
let munge (b:bool) (x:int) : ?? =
  if not b then
    string_of_int x
  else
    "hello"
;;

let y = 17;;
```

```
munge (y > 17) : ??

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

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

Rule for type-checking functions

Example:

```
let munge (b:bool) (x:int) : ?? =
  if not b then
    string_of_int x
  else
    "hello"
;;

let y = 17;;
```

```
munge (y > 17) : ??

munge true (f (munge false 3)) : ??
  f : string -> int

munge true (g munge) : ??
  g : (bool -> int -> string) -> int
```

One key thing to remember

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

A -> B -> C -> D -> E -> 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 -> C -> D -> E -> F (if a1 : A)

f a1 a2 : C -> D -> E -> F (if a2 : B)

f a1 a2 a3 : D -> E -> F (if a3 : C)

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

OUR FIRST* COMPLEX DATA STRUCTURE! 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

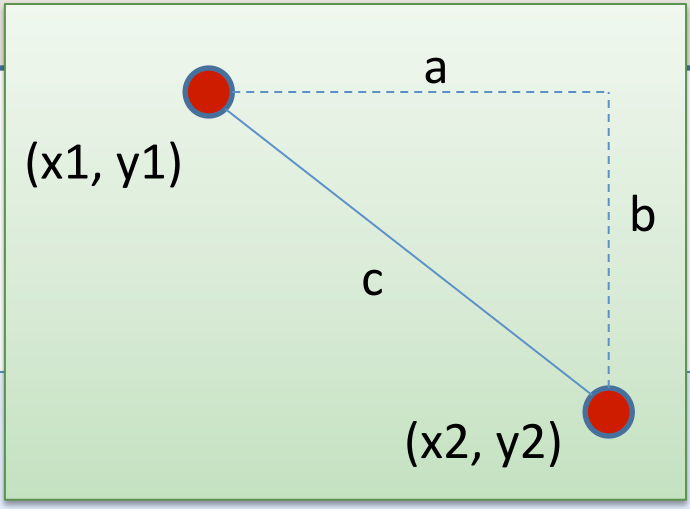
if e1 : t1 * t2 then
x1 : t1 and x2 : t2
inside the expression e2

```
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

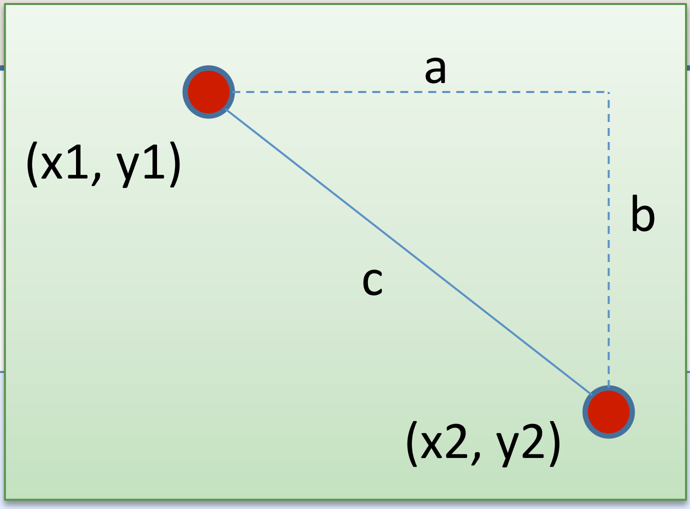
Writing Functions Over Typed Data

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.

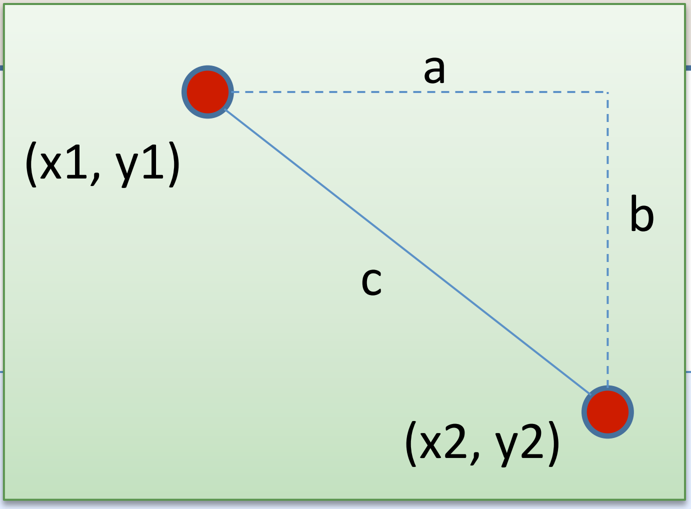
Distance between two points



a type abbreviation

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

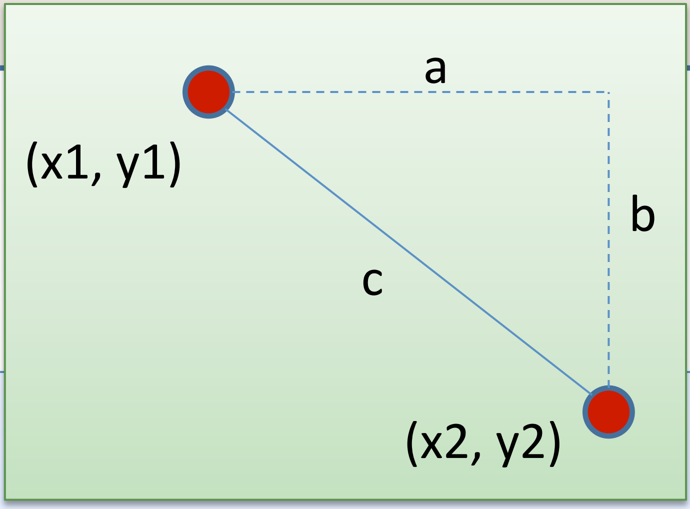
Distance between two points




```
type point = float * float

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



;;
```

write down function name
argument names and types

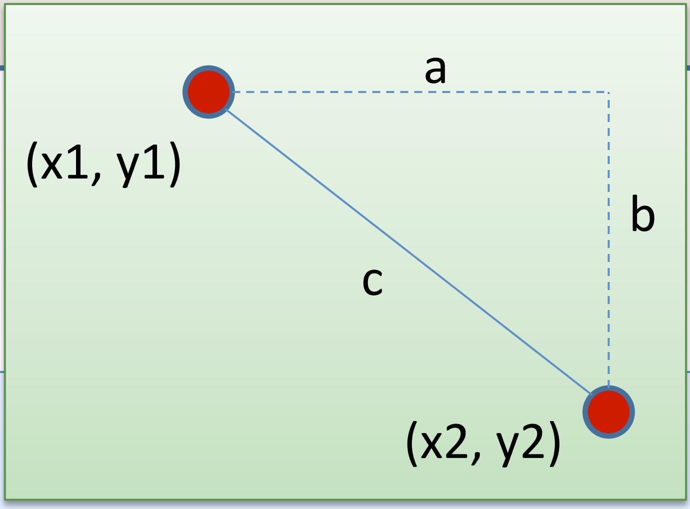
Distance between two points



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 =
```

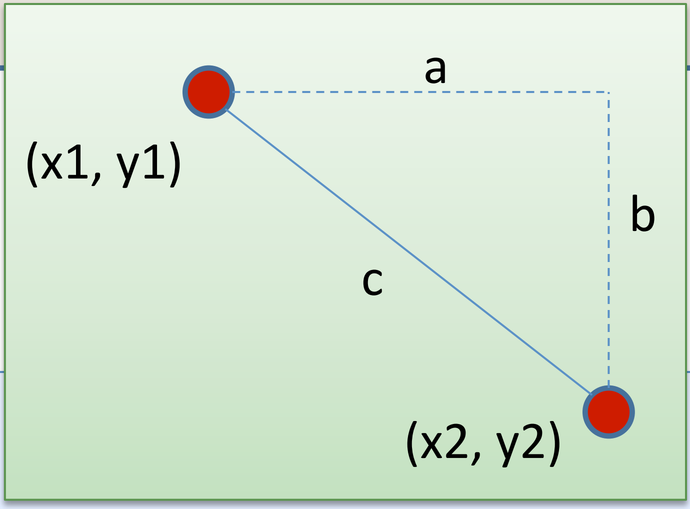
Distance between two points




```
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

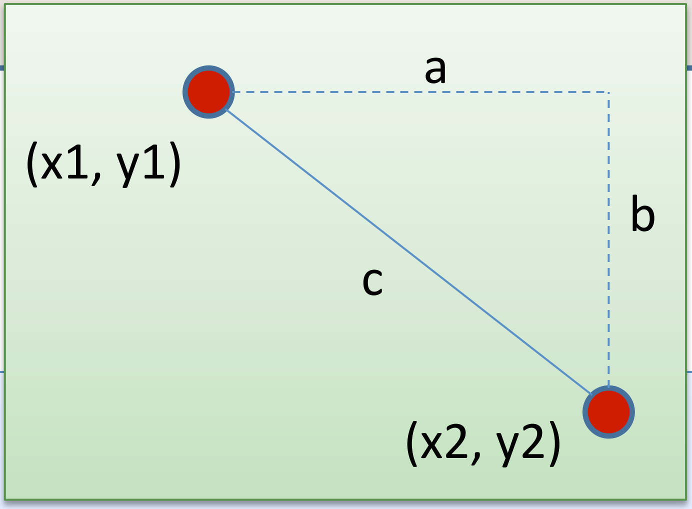



```
type point = float * float

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

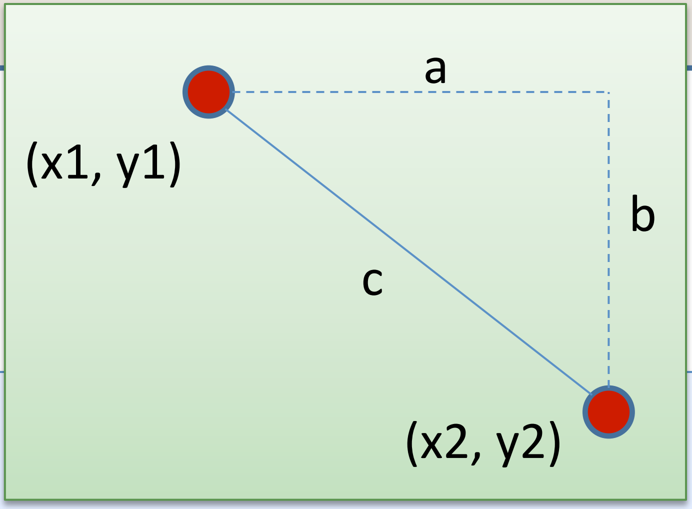
Distance between two points




```
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 (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;;
```

testing

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
   - the *argument types* suggest how to do it
5. Build new output values
   - the *result type* suggest 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
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

END